AFRL-ML-WP-TR-2000-4147

SIMULATION ASSESSMENT VALIDATION ENVIRONMENT (SAVE)

Computer Software Product End Item

Paul Cole, Bob Bassett, Marcia Herndon, Paul Collins, Kathy Jacobson

Lockheed Martin Tactical Aircraft Systems
1 Lockheed Blvd.
Fort Worth, TX 76108

October 2000

Final Report for the Period November 1997 – August 2000

Materials & Manufacturing Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson Air Force Base, Ohio 45433-7750

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASC/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

JAMES W. POINDEXTER
Project Engineer
Information Integration Team
Advanced Manufacturing Enterprise Branch

PATRICK E. PRICE
Leader
Information Integration Team
Advanced Manufacturing Enterprise Branch

WILLIAM E. RUSSELL
Chief
Advanced Manufacturing Enterprise Branch
Manufacturing Technology Division

"If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFRL/MLMS, Bldg. 653, 2977 P St., Suite 6, W-PAFB, OH 45433-7739 to help us maintain a current mailing list."

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 2000	Final 11/97 - 08/00

4. TITLE AND SUBTITLE
Simulation Assessment Validation Environment (SAVE)
Computer Software Product End Item

5. FUNDING NUMBERS
C F33615-95-C-5538
PE 63800F
PR 2025
TA 50
WU 03

6. AUTHOR(S)
Paul Cole, Bob Bassett, Marcia Herndon, Paul Collins, Kathy Jacobson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Lockheed Martin Tactical Aircraft Systems
1 Lockheed Blvd.
Fort Worth, TX 76108

8. PERFORMING ORGANIZATION REPORT NUMBER
JRES010452

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Materials & Manufacturing Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson AFB, OH 45433-7750
POC: James W. Poindexter, AFRL/MLMS, (937) 904-4351

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-ML-WP-TR-2000-4147

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution is Unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The JSF Simulation Assessment Validation Environment (SAVE) program provides the capability to assess the manufacturing impacts of both product and manufacturing process design decisions. By integrating Commercial Off The Shelf (COTS) modeling and simulation tools into a *seamless virtual environment*. SAVE allows design teams to develop and verify new affordable aircraft concepts before developing expensive hardware.

This document is intended to provide a programmer's and implmenter's view of information about the simulation Assessment Validation Environment (SAVE) system. It is divided into a series of chapters and appendices for ease of access to the desired information. The core chapters discuss the details of the SAVE architecture and tool integration specification, the approach to developing SAVE compliant tools, software developed to demonstrate and validate SAVE, and recommended enhancements and extensions for future development. The appendices provide s specific reference information that is invaluable to developers of SAVE compliant software and to those tasked with implementing SAVE systems.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Virtual Manufacturing (VM), Joint Advanced Strike Technology (JAST), Joint Strike Fighter (JSF), Lean Implementation.	252
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

NOTICES

Deneb Robotics, Inc.

Quest® - Registered trademark of Deneb Robotics, Inc.

IGRIP® - Registered trademark of Deneb Robotics, Inc.

ERGO® - Registered trademark of Deneb Robotics, Inc.

Symix Corporation

Factor/AIM® - Registered trademark of Symix Corporation

Cognition Corporation

Cost Advantage™ - trademark of Cognition Corporation

SAIC® - Registered trademark of Science Applications International Corporation

CATIA® - Registered trademark of Dassault Systemes

Dassault Systemes

SGI ™ - Trade Mark of Silicon Graphics, Inc.

Microsoft ®- Registered trademark of MicroSoft Corporation

Windows® ™ - Registered trademark or Trade Mark of MicroSoft Corporation

Windows NT® ™ - Registered trademark or Trade Mark of MicroSoft Corp.

JMCATS

AIX ™ - Trade Mark of International Business Machines

IBM® - Registered trademark of International Business Machines

UNIX® - Registered trademark of UNIX System Labs, Inc.

Collabora®- Registered trademark of Netscape, Inc.

Engineering Animation, Inc.

General Notice: Some of the product names used herein have been used for identification purposes only and may be trademarks or registered trademarks of their respective manufacturers.

ABSTRACT

The JSF Simulation Assessment Validation Environment (SAVE) Program provides the capability to assess the manufacturing impacts of both product and manufacturing process design decisions. By integrating Commercial Off-The-Shelf (COTS) modeling and simulation tools into a *seamless virtual environment*, SAVE allows design teams to develop and verify new affordable aircraft concepts before developing expensive hardware.

The SAVE infrastructure utilizes a CORBA-based shared Data Model and Work Flow Manager and a commercial *Electronic Collaborative Design Notebook* to integrate a suite of six commercial manufacturing tools which include schedule, factory, assembly, dimensional variability, cost, and risk simulations. In the future, other tools may be added by developing simple SAVE-compliant CORBA wrappers, and SAVE will be available to extend to other problem domains such as operations and support simulations to assess life-cycle issues.

SAVE expects to achieve significant cost savings for the JSF Program by providing integrated design teams the capability to quickly perform "what-if" studies and accurately define a product's cost and risk early in the design process. While the SAVE initiative is vital to achieving the affordability goals of the Joint Strike Fighter, the implementation of SAVE in other design and manufacturing environments has the potential to generate equally impressive cost savings.

This document, The Software Product End Item, is intended to provide a programmer's and implementer's view of information about the Simulation Assessment Validation Environment (SAVE) system. It is divided into a series of chapters and appendices for ease of access to the desired information. The core chapters discuss the details of the SAVE architecture and tool integration specification, the approach to developing SAVE-compliant tools, software developed to demonstrate and validate SAVE, and recommended enhancements and extensions for future development. The appendices provide specific reference information that is invaluable to developers of SAVE-compliant software and to those tasked with implementing SAVE systems.

FOREWORD

Commercial industry is leading the way in implementing the use of modeling and simulation tools to reduce product cost, time to market, etc. The use of these tools results in leaner systems that are more competitive in the global market. The emphasis within commercial industry is not only to stay in business, but become more profit conscious. Many companies are seeing declining revenues and higher profits. How is this possible? They have found ways to reduce cost, in other words make their products more affordable. This occurs in many ways from streamlined production, to the rapid introduction of new products, to strategic partnering, including outsourcing or cosourcing.

The ability to simulate manufacturing operations, prior to actual production, is having a profound impact on product and process design decision making. Commercial simulation tools have matured rapidly in recent years, but their use is still somewhat limited by the lack of integration among sets of tools to evaluate cost, schedule, and risk. The SAVE program was initiated to address this required integration.

The concept of affordability is a central theme in the Joint Strike Fighter (JSF) program. This is seen in the genesis of the program by combining three products into one to leverage affordability by streamlining development and production cost. In concept, all three derivative aircraft are designed and manufactured jointly, with the exception of parts that are affected by customer specific requirements (e.g., Navy, carrier based models, require additional structural enhancements for the undercarriage). The development activity is characterized by a single design and manufacturing effort that encompasses all common parts with a split near the end to handle customer specific requirements. The net result will be an affordable fighter realized through the leveraging of common design and manufacturing efforts.

Affordability is further explored within the Manufacturing and Producibility IPPT through the sponsorship of six key initiatives. These include:

- JSF Manufacturing Capabilities Tool Set (JMCATS) - Developing a tool set for analyzing manufacturing risk and capabilities with traceability back to basic product requirements and functions.

- JSF Manufacturing Demonstration Program (JMD) - Developing an IPPT process with supporting tools to assess manufacturing cost directly from CAD data bases and to collect manufacturing information needed to drive cost engines.

- Virtual Manufacturing Fast Track Program - An initial JSF demonstration to show the usefulness of virtual manufacturing using an integrated environment of available software design and manufacturing tools.

- Ribbonized, Organized, Integrated (ROI) Wiring Program - A JSF demonstration to show the potential weight and cost savings using an ROI wiring architecture in a tactical aircraft.

- Manufacturing Affordability Development Program (MADP) - A JSF Government Team survey of twelve companies at seventeen facilities to identify pockets of manufacturing and producibility successes which demonstrated affordability potential for the remainder of the industry.

- Simulation Assessment Validation Environment Program (SAVE) - Develop and demonstrate a virtual manufacturing environment through the integration of a set of simulation, modeling and analysis tools.

Combined these programs are estimated to achieve between 12%-20% reduction in life cycle cost through demonstration and implementation of improved processes and tools which reflect manufacturing considerations early in design.

These programs were identified as a result of the 1994 Government led Lean Forum Workshop. The consensus topics of critical enabling technologies for JSF affordability identified from this workshop were, Integrated Design and Cost; Modeling and Simulation; Teaming; Factory Operations; and Design for Quality and Producibility. The results of this workshop have led to the JSF sponsored programs listed above. The SAVE program addresses the consensus topic of Modeling and Simulation.

The SAVE program is the integration of best of breed commercial off the shelf tools that support the generation and analysis of data needed to make affordability based decisions. This leads to an ability to perform cost/performance trade studies, thereby enabling the treatment of cost as an independent variable by making cost clearly quantified as design requirements and decisions are made. The end result will be a new set of commercially available capabilities that can support the entire JSF customer, prime, team, supplier and user base. SAVE provides the ammunition to drive affordability at all levels in the program. The SAVE program is estimated to contribute 1%-2% of the life cycle cost savings listed above.

This report documents the programmer and implementers information on the software that was developed by the SAVE contract.

PREFACE AND ACKNOWLEDGMENTS

This report provides information on the activities and results developed during the time period of November 1997 through August 2000 and has been prepared for AFRL/MLMS as CDRL number A0011 for contract number F33615-95-C-5538.

The report consist of seven major sections:

1. Introduction and scope of this document

2. Discussion of the SAVE interface specification. This section describes the basic approach and philosophy behind the SAVE integration strategy. Formal specifications are provided in Common Object Request Broker Architecture (CORBA) Interface Definition Language (IDL) in Appendix E.

3. The integration and development process that is used to produce SAVE-compliant simulation tools and elements of the infrastructure are described.

4. Detailed discussions of each of the major SAVE-developed software programs are provided.

5. A series of four knowledge-based cost models have been developed for the SAVE-compliant version of Cognition Corporation's CostAdvantage (CA) tool. The SAVE-developed CostLink between CA and the CATIA CAD tool is also described.

6. A list of recommended enhancements to the SAVE system is presented.

7. A series of six appendices contain detailed specifications, sample code listings, SAVE vendor product descriptions, tool specific input / output capabilities, and a description of the directories that are on the associated CDROM.

The following individuals are acknowledged for their work and dedication in achieving the successful results of Phase II Interim activities:

Paul Cole	Program Manager
Bob Bassett	Architecture/Infrastructure IPPT Lead
Marci Herndon	Tool Integration IPPT Lead
Paul Collins	Demonstration/Technology Transfer IPPT Lead
Kathy Jacobson	Enhanced Design Cost Integration IPPT Lead

LIST OF ACRONYMS AND ABBREVIATIONS

ABC - Activity Based Costing
ADAM - Affordable Design And Manufacturing
AFMC - Air Force Materiel Command
AFRL - Air Force Research Laboratory
AIC - Artificial Intelligence Center
AIMS - Agile Infrastructure for Manufacturing Systems
ANSI - American National Standards Institute
API - Application Programming Interface
ASC - Aeronautical Systems Center
Assy - Assembly
ASURE - Analysis System for Uncertainty and Risk Estimation
BAFO - Best and Final Offer
BCL - Batch Control Language
BOM - Bill of Material
CA - Cost Advantage
CAD - Computer Aided Design
CATIA - Computer Aided Three Dimensional Interactive Application
CAx - Computer Aided (Anything)
CDA - Concept Demonstrator Airplane
CDE - Common Desktop Environment
CDF - Common Data File
CDP - Concept Demonstrator Program
CDRL - Contract Data Requirements List
CE - Concurrent Engineering
CER - Cost Estimating Relationship
COM - Common Object Model
CORBA - Common Object Request Broker Architecture
COSE - Common Open System Environment
COTS - Commercial Off the Shelf
CSA - Close Support Aircraft
CSC - Computer Science Corporation
CTC - Concurrent Technologies Corporation
DARPA - Defense Advanced Research Project Agency
DB - Database
DDI - Decision Dynamics Incorporated
DEAM - Design Engineering Analysis Model
DFM - Design for Manufacturability
DICE - DARPA Initiative on Concurrent Engineering
DID - Data Item Description
DLL - Dynamic Link Library
DMC - Defense Manufacturing Conference
DNS - Domain Name Service
DoD - Department of Defense

DOE - Design of Experiments
E&MD - Engineering and Manufacturing Development
EAF - Estimate Adjustment Factor
EAI - Engineering Animation, Incorporated
EBOM - Engineering Bill of Materials
ECRC - Electronic Commerce Resource Center
EDN - Electronic Design Notebook
EMD - Engineering & Manufacturing Development
ERGO - Ergonomics Option
ERP - Enterprise Resource Planning
FA - Factor AIM
Fab - Fabrication
GD&T - Geometric Dimensioning & Tolerancing
GDT - Geometric Dimensional Tolerancing
GE - General Electric
GIFF - Graphical Interface File Format
GII - Graphic Interactive Interface
GIT - Georgia Tech Institute of Technology
GRCI - General Research Corporation International
GSL - Graphics Simulation Language
GUI - Graphical User Interface
H/W - Hardware
HTML - Hypertext Markup Language
I/F - Interface
I/O - Input/Output
IBAM - Industrial Base Analysis Model
IBM - International Business Machines
ID - Identification
IDL - Interface Definition Language
IGES - Initial Graphics Exchange Specification
IGRIP - Interactive Graphics Robot Instruction Program
IIOP - Internet Inter-ORB Protocol
IP - Internet Protocol
IP/PD - Integrated Product/Process Development
IP/PT - Integrated Product/Process Team
IPAM - Industrial Production Analysis Model
IPD - Integrated Product Development
IPPDB - Integrated Product Process Database
IPPT - Integrated Product and Process Team
IPT - Integrated Product Team
IRS - Interface Requirements Specification
ISO - International Standards Organization
JAR - JAVA Application Resource
JAST - Joint Advanced Strike Technology
JDK - JAVA Development Kit
JMCATS - JSF Manufacturing Capabilities Assessment Tool Set

JMD - JSF Manufacturing Demonstration
JPO - Joint Program Office
JSF - Joint Strike Fighter
KB - Knowledge Base
KC - Knowledge Center
LAI - Lean Aerospace Initiative
LCAM - Life Cycle Analysis Model
LCC - Life Cycle Cost
LHS - Left Hand Side
LM - Lockheed Martin
LMMS - Lockheed Martin Missiles and Space
LMSW - Lockheed Martin Skunk Works
LRIP - Low Rate Initial Production
MADE - Manufacturing Automation and Design Engineering
MADP - Manufacturing Affordability Development Program
MBOM - Manufacturing Bill of Materials
MDF - Metadata File
ME - Manufacturing Engineer
MECE - Multimedia Environment for Concurrent Engineering
Mfg - Manufacturing
Mgt - Management
MRP - Manufacturing Resource Planning
MS - Microsoft
N/C - Numerical Control
NB - Net Builder
NC - Numerical Control
NFS - Network File System
NIIIP - National Industrial Information Infrastructure Protocol
NIST - National Institute of Standards and Technology
ODBC - Open Data Base Connectivity
OMG - Object Management Group
OO - Object Oriented
ORB - Object Request Broker
OSHA - Operational Safety and Health Administration
OTF - Operational Task Force
P&W - Pratt and Whitney
PC - Personal Computer
PCA - Physical Configuration Audit
PCM - Production Cost Model
PDM - Product Data Manager
PDR - Preliminary Design Review
PEO - Program Engineering Office
PIE - Probabilistic Inference Engine
POOP - Plain Old Orbix Protocol
QM - Query Manager
QUEST - Queuing Event Simulation Tool

R&D	- Research & Development
RADM	- Rear Admiral
RASSP	- Rapid Prototyping of Application Specific Signal Processors
RDB	- Relational Database
RDE	- RASSP Design Environment
RFP	- Request for Proposal
RHS	- Right Hand Side
ROI	- Return on Investment
ROI	- Ribbonized, Organized, Integrated Wiring
RPC	- Remote Procedure Call
S/W	- Software
SAIC	- Science Applications International Corporation
SAVE	- Simulation Assessment Validation Environment
SBD	- Simulation Based Design
SCRA	- South Carolina Research Authority
SDAI	- Software Data Access Interface
SDE	- SAVE Design Environment
SDM	- SAVE Data Model
SDR	- System Design Review
SGI	- Silicon Graphics Incorporated
SLMC	- Sanders a Lockheed Martin Company
SMC	- Systems Modeling Corporation
SQL	- Structured Query Language
SRR	- Scrap, Rework, and Repair
SS	- Source Selection
STAM	- Strategic Technologies Analysis Model
STL	- SteroLithography
T/BAB	- Technical/Business Advisory Board
TBD	- To Be Determined
TCP/IP	- Transmission Control Protocol/Internet Protocol
TDM	- Technical Data Management
UG	- UniGraphics
USAF	- United States Air Force
USC	- University of Southern California
VM	- Virtual Manufacturing
VP	- Virtual Prototyping
VSA	- Variability Simulation Analysis
WFM	- Work Flow Manager
WL	- Wright Laboratory
WPAFB	- Wright Patterson Air Force Base

1.0 Introduction and Scope

1.1 Introduction - Rapid Growth in the Desire to Apply Virtual Manufacturing

When Concurrent Engineering (CE) burst upon the scene in the mid-1980s, acceptance of its concepts grew rapidly. The central precept of CE is the use of multidisciplinary teams representing all stakeholders of the design, manufacturing and business processes. Each team focuses on the combined problems of product and process development, and strives to eliminate the "over-the-wall" hand-off of data from one organization to the next. Early adopters of the CE approach demonstrated significant improvements in product cost, quality, and time to market. Early consideration of the manufacturing impacts of design decisions clearly results in identifying better designs, early identification of problems, and reduced scrap, rework, repair, and redesign. Application of CE, often called Integrated Product Development, is now widespread.

As might be expected, the cultural impediments to implementing CE, particularly in large design teams, can be significant. One issue that arises is the varying levels of detail that different team members can bring to bear on a design in the early phases of development. As design concepts are developed teams must balance the impacts on a range of performance considerations, cost, producibility, schedule and risk. Often, the traditional analysis disciplines (performance, weight, structural strength, etc) can make clear, quantified, strong cases for impacts in their areas. Producibility, cost, schedule, and risk have tended to be more subjective and based on experience rather than analysis. Difficult design decisions tend to be made in favor of the cleanly quantified issue - it won't perform, it weighs too much, it will break, etc. Serious manufacturing issues become concrete at a later, costlier phase of a project - during manufacturing.

Recognition of these shortcomings in CE, fueled by the almost explosive growth in affordable computer power, is leading companies to apply the tools of virtual manufacturing.

Virtual Manufacturing (VM) is the integrated use of design and production models and simulations to support accurate cost, schedule and risk analysis. These modeling and simulation capabilities allow design teams to rapidly and accurately determine production impact of product/process alternatives through detailed simulation of important processes. The simulation results allow manufacturing impacts to be more evenly balanced against traditional analyses.

The potential for VM tools to significantly improve affordability and reduce cycle times is widely accepted, but the potential has not been fully achieved. Many commercial manufacturing simulation tools with excellent capabilities exist on the market today. Although, many of these tools rely on similar types of data, differences in internal storage structures and nomenclature have prevented easy tool to tool data integration. Often, large amounts of data must be reentered, at considerable time and expense, to accommodate these differing formats. Some point-to-point solutions do exist between specific tools, but as the number of tools grows, this integration solution becomes unmanageable, and the benefits from using an integrated tool suite go unrealized.

1.2 The Need for SAVE

The Simulation Assessment Validation Environment (SAVE) program, just completed by Lockheed Martin through funding from the Joint Strike Fighter Program Office, addressed these limitations by developing and implementing an open architecture environment to integrate a representative suite of manufacturing modeling and simulation tools. SAVE also demonstrated this integrated simulation capability to significantly reduce product life cycle costs.

The initial phase of the program, completed in August 1996, established a core tool suite integrated via the Defense Advanced Research Projects Agency (DARPA) developed Rapid Prototyping of Application Specific Signal Processors (RASSP) architecture. The core tool suite incorporated commercial CAD, factory simulation, assembly simulation, schedule simulation, cost and risk modeling capabilities.

During Phase II, the SAVE team developed a Common Object Request Broker Architecture (CORBA) based approach to tool integration which provides a solid foundation for ultimate production use and commercialization of SAVE. The CORBA-based infrastructure now includes the SAVE Common Data Model, a Work Flow Manager, and a Query System for interactive access to the Data Model. In addition, commercially available dimension and tolerance simulation capabilities have been added to the VM environment. An Electronic Collaborative Design Notebook is considered essential to SAVE, and although it was not developed under the contract, a commercially available web-based product was used for Phase II.

1.3 Objectives of SAVE

In recent years, manufacturing modeling and simulation software has seen increased use throughout industry. Rapid advances in computing hardware and software now allow accurate simulations of complex processes. Computer graphics provide Integrated Product/Process Teams (IPPT) with the means to effectively understand the results of these simulations and make critical design and manufacturing decisions, without resorting to costly physical prototypes.

Growth in the use of virtual manufacturing tools has only been limited by the costly, manual transfer of data among the set of simulation tools. Typically, a design team will use a 2-D or 3-D CAD package for design. The team will then assess the manufacturing impact of product and process decisions through use of a set of virtual manufacturing tools to assess cost, schedule, and risk. The tool capabilities typically include:

- Process planning
- Dimension and tolerance analysis
- Schedule simulation
- Assembly simulation

- Factory simulation
- Ergonomic simulation
- Feature-based costing

These tools use much of the same data as input, but each requires different internal data formats. Manual reformatting and reentry of these data are prohibitively costly. The vision of SAVE is to provide a system for integration of simulation codes into an efficient, easy-to-use capability that rapidly assesses the manufacturing impacts (cost, schedule, and risk) of product and process design decisions.

1.4 Overview of SAVE Technical Approach

In order to understand the use of the SAVE Virtual Manufacturing Environment, it is necessary to first gain an understanding of the basics of the technical approach to creating the environment. The two primary elements of SAVE include the simulation tool integration and the tool execution and management infrastructure. The integration allows tools to share common data without concern for their computer platform, the location of that computer, or the language used to program the tool. The execution and management infrastructure facilitates communication, management, and access among the IPPT members using the system.

The components of the SAVE environment and their interfaces within the system are shown in Figure 1-1. Together, these components provide an integrated Virtual Manufacturing capability.

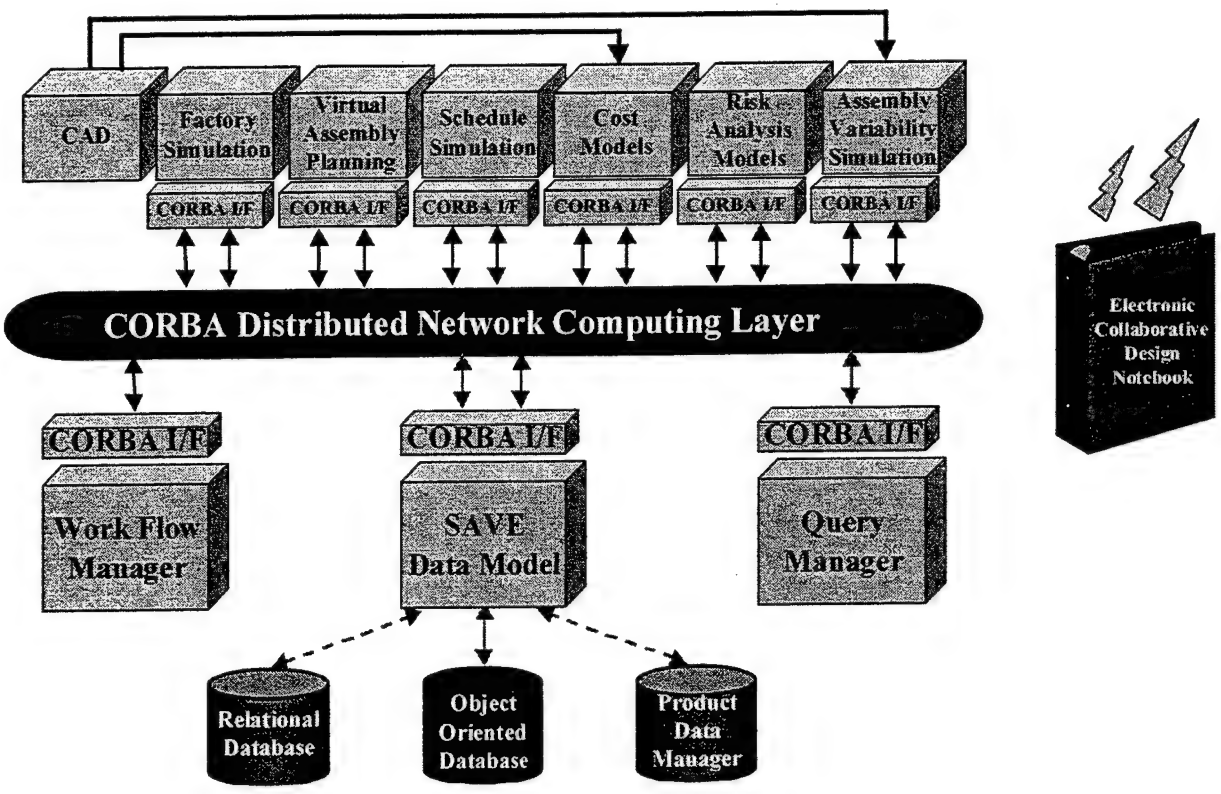

Figure 1-1: SAVE Technical Approach

The concept starts with classes of tools that are generally available in the commercial market. These tool classes are identified across the top of Figure 1-1. They include the following:

- CAD
- Factory Simulation
- Virtual Assembly Planning
- Schedule Simulation
- Cost Modeling
- Risk Analysis
- Assembly Variability Simulation

At the heart of the infrastructure is the SAVE Data Model (SDM). It represents the data that is common, or shared, among the tools and provides the contract for data exchange among the various software tools. A graphical overview of the elements of the SDM is shown in Figure 1-2. The definitions and makeup of the data in the SDM are described in detail in Appendix C. The SDM is designed so that it can be implemented with connections to various data storage devices. This allows production implementations of SAVE to access the data wherever it resides within the enterprise. For example, tooling data may reside in the Product Data Management System, while material data may be maintained in a separate relational database.

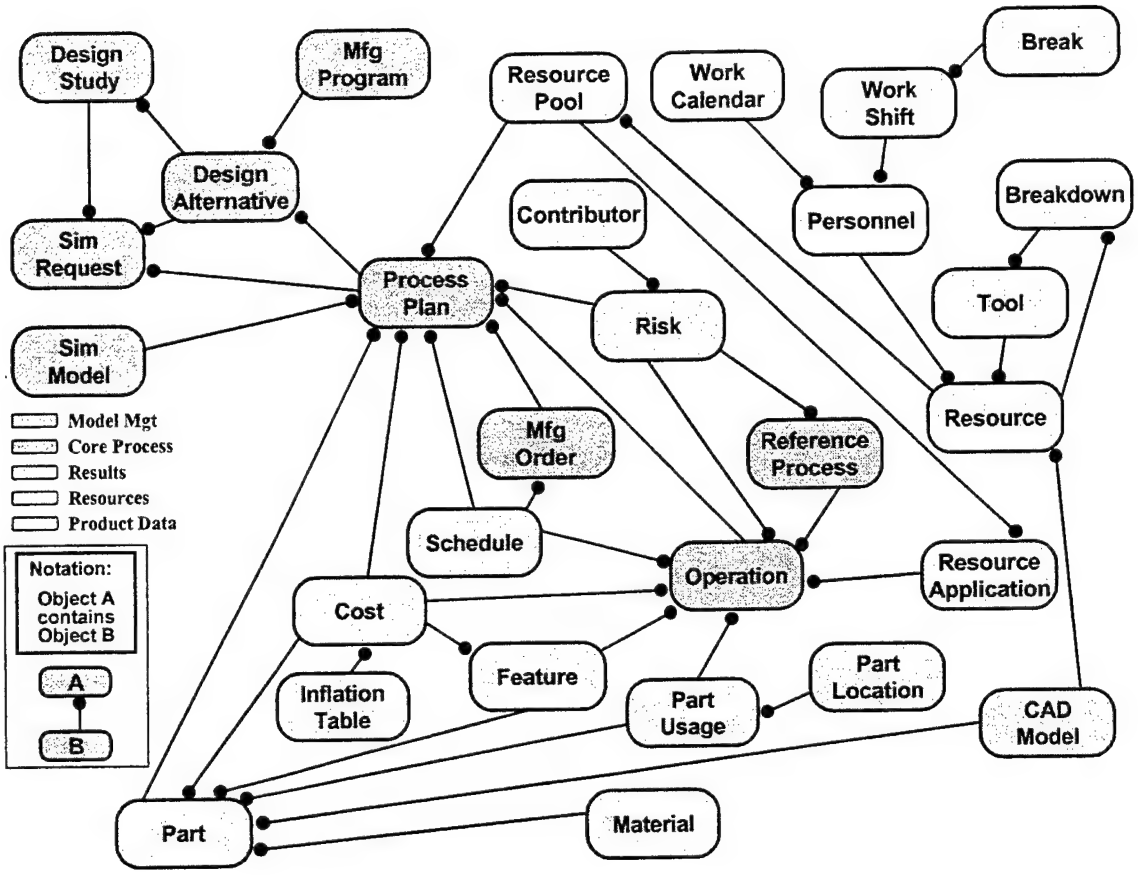

Figure 1-2: The SAVE Data Model

The SAVE architecture also contains a Work Flow Manager (WFM) that provides graphical process modeling and tool execution. More details in the use of the WFM are provided in Appendix L. In order to provide visibility into the information contained within the SDM, a Query Manager (QM) application was developed by the SAVE team. This application, described in Appendix M, provides the capability to browse, create, modify and delete information in the SDM directly.

All of these components communicate with one another through Common Object Request Broker Architecture (CORBA) interfaces that adhere to the specifications of the SDM and WFM. The use of CORBA in the SAVE architecture provides two primary benefits:

1. Software vendors develop a single interface for data exchange with other tools as opposed to the point to point interfaces that would be required without the use of a common data model.

2. Implementations allow data storage locations to be defined by the deployment sites, not the software development team.

In addition to the CORBA-compliant portions of the SAVE architecture, SAVE contains two additional features. An electronic collaborative design notebook allows users to

communicate with one another and share information that is not part of the common data. This notebook may also be used to collect decision-making history. Three-dimensional CAD data is a key element of many simulation models. The SAVE environment provides a direct link from most major CAD systems for extracting pertinent feature data to both the cost analysis tool and the assembly variability simulation tool. Currently, the CAD data is translated for use in the other tools, but direct links are quickly becoming available.

1.5 Scope Of This Document

This document, The Software Product End Item, is intended to provide a programmer's and implementer's view of information about the Simulation Assessment Validation Environment (SAVE). It is divided into a series of chapters and appendices for ease of access to the desired information. The core chapters discuss the details of the SAVE architecture and tool integration specification, the approach to developing SAVE-compliant tools, software developed to demonstrate and validate SAVE, and recommended enhancements and extension for future development. The appendices provide specific reference information that is invaluable to developers of SAVE-compliant software and to those tasked with implementing SAVE systems.

The following list provides a summary of topics and their location within the document:

Chapter 1	Introduction and Scope
Chapter 2	Interface Specification
Chapter 3	Integration and Development Process
Chapter 4	SAVE-Developed Software Description
Chapter 5	Recommended Enhancements
Chapter 6	Cost Model Development
Appendix A	SAVE Deliverable Software Listing
Appendix B	SAVE Vendor Product Descriptions
Appendix C	Tool Specific Input/Output Capabilities
Appendix D	Sample Code
Appendix E	SAVE CORBA IDL

2.0 Interface Specification

This interface specification contains detailed descriptions of the concepts employed in the SAVE architecture and of the requirements for integrating simulation software tools into the environment. Tool interfaces into SAVE are comprised of two components: a data-sharing client and a work flow server. These interfaces provide the mechanism for manufacturing simulation tools to work within the Virtual Manufacturing (VM) environment.

2.1 Overview of the SAVE Architecture

The SAVE architecture, shown in Figure 2-1, contains several components that, when combined, provide a VM environment through the integration and data sharing among commercially available tools. The architecture concept starts with the tool classes that are necessary in a VM environment. These tools are identified across the top of Figure 2-1 and are described in more detail in Section 2.2. At the heart of the infrastructure is the SAVE Data Model (SDM). It represents the data that is common, or shared, among the tools and provides the contract for data exchange among the various software tools. The SDM is designed so that it can be implemented with connections to various data storage devices. The SDM is described in detail in Section 2.3 of this document. The SAVE architecture also contains a Work Flow Manager (WFM) that provides graphical process modeling and tool execution. The interface between the WFM and the simulation tools, called a Simulator, is described in Section 2.4. In order to provide visibility into the information contained within the SDM, a Query Manager (QM) application was developed. This application, described in Section 2.5, provides the capability to browse, create, modify and delete information in the SDM directly and independent of any of the integrated tool classes. All of these components communicate with one another through Common Object Request Broker Architecture (CORBA) interfaces that adhere to the specifications of the SDM and WFM. The use of CORBA in the SAVE architecture is discussed in Section 2.6. In addition to the CORBA-compliant portions of the SAVE architecture, SAVE contains an electronic collaborative design notebook. This notebook allows users to communicate with one another and share information that is not part of the common data (SDM).

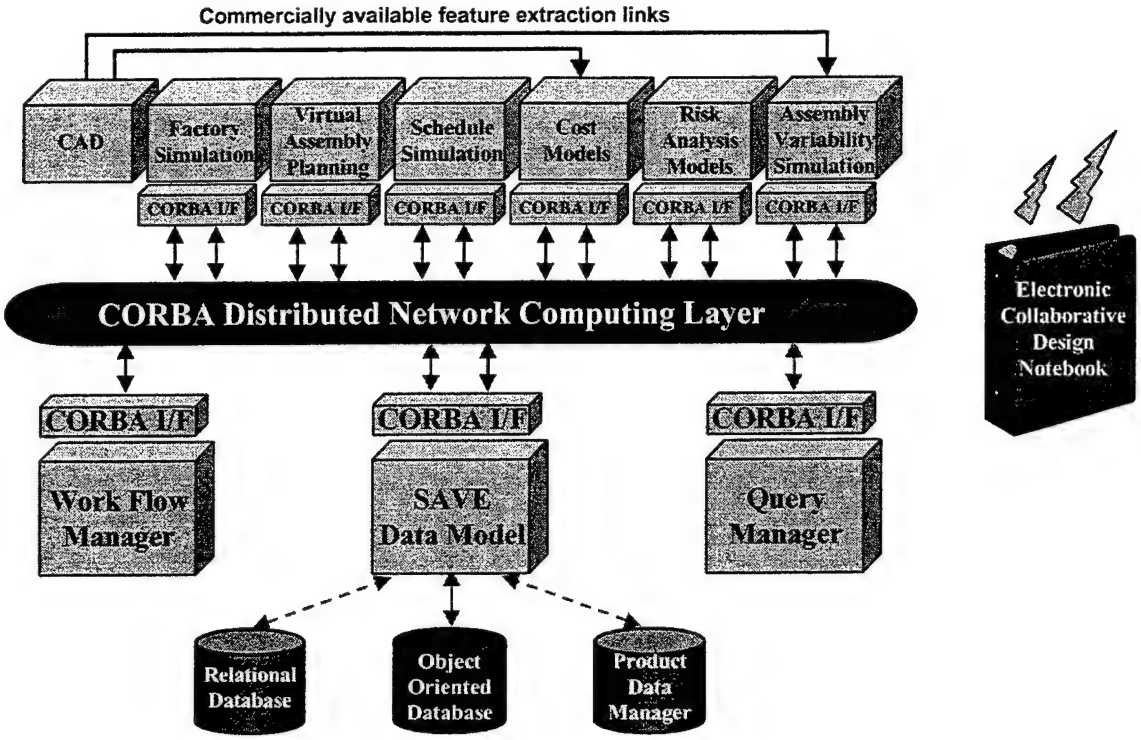

Figure 2-1 CORBA Approach to Tool Integration

2.2 SAVE Tool Classes

Specific vendor tools considered best-in-class are being integrated into the infrastructure as part of the formal SAVE program; however, the SAVE infrastructure is a flexible, open architecture that allows easy integration of new tools into the overall system. The SAVE tools fall into the following categories: CAD, factory simulation, virtual assembly planning, schedule simulation, cost modeling, risk analysis, and assembly variability simulation.

The data represented in the SDM is comprised of information shared among manufacturing simulation tools and information necessary for assessing the outcome of the tool's simulation. Tools may share common input information or output from one tool may be input for another. There are seven categories of tools being integrated in the SAVE environment. Each tool provides a different view of the manufacturing simulation.

2.2.1 CAD

The CAD tool provides the geometric definition of the product and is typically populated by a user. The information produced as a part of the CAD model is useful for any category of simulation tool that needs the product representation. Figure 2-1 shows two existing direct interfaces from the CAD tool to Cost Models and Assembly Variability Simulation. These interfaces exist for four major CAD tools and allow CAD to be loosely coupled to SAVE. In future versions of SAVE, the CAD tool could be wrapped

to output cost and tolerance feature data directly into the SAVE Data Model. Figure 2-2 shows the top-level interfaces for a typical CAD tool.

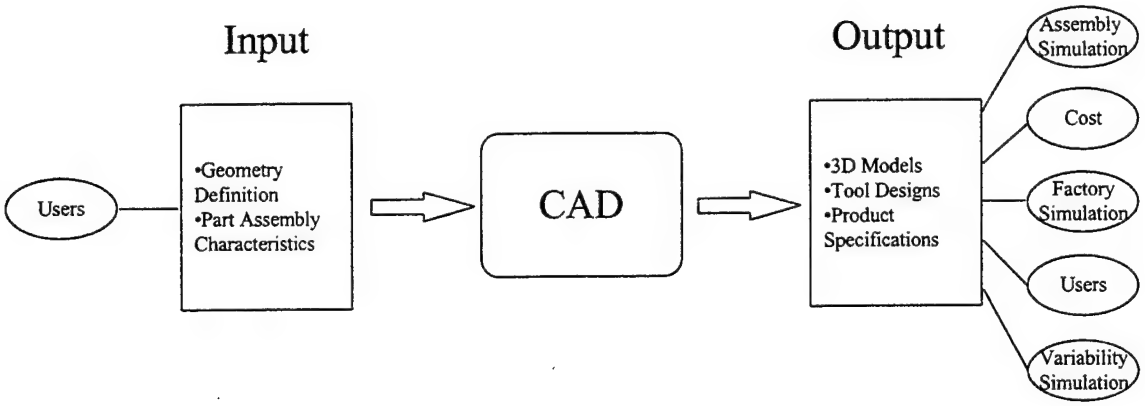

Figure 2-2 CAD Tool Interfaces

2.2.2 Factory Simulation

The factory simulation tool uses the process plan information as well as factory layouts to provide factory planning including throughput, layout, and resource allocation. Inputs are provided by a variety of sources, including the schedule simulation and assembly planning tools. The data provided by this class of tool is used in estimating cost, schedule, and risk for the design alternative being simulated. Figure 2-3 shows the top-level interfaces for a typical factory simulation tool.

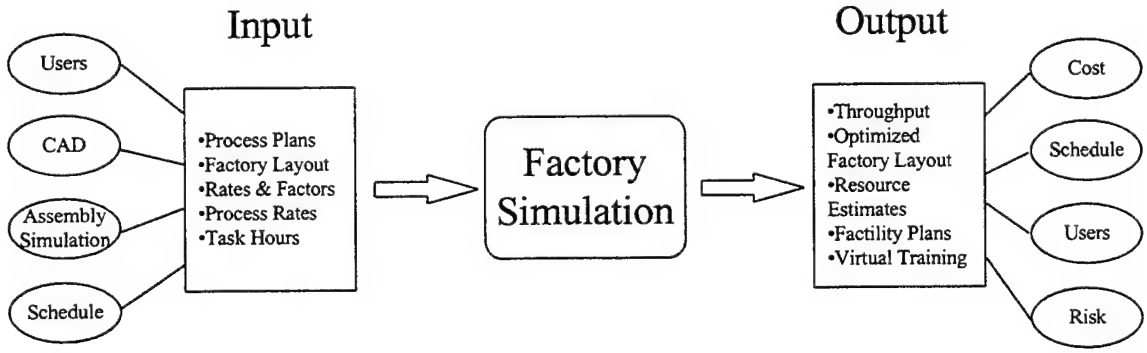

Figure 2-3 Factory Simulation Tool Interfaces

2.2.3 Virtual Assembly Planning

The virtual assembly planning tool uses associated product models and tool designs to produce assembly work instructions (or process plan) along with the hours associated with each task. This information is used in factory simulation, scheduling, and cost estimation. Figure 2-4 shows the top-level interfaces for a typical virtual assembly planning tool.

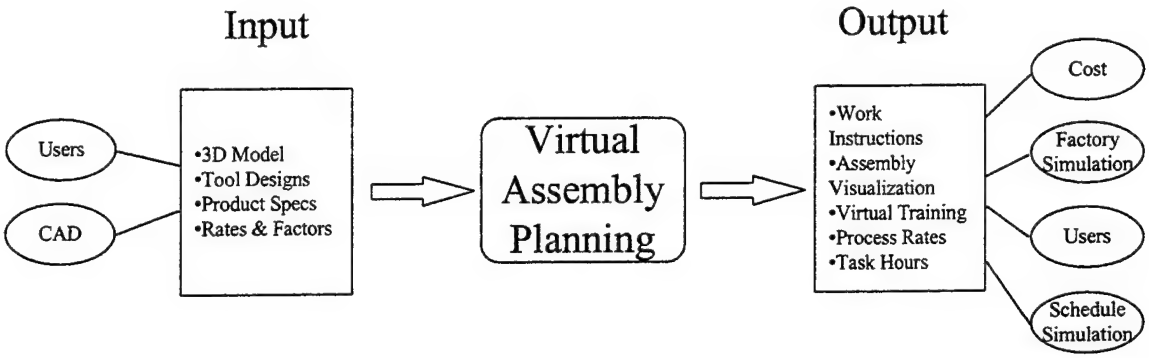

Figure 2-4 Virtual Assembly Planning Tool Interfaces

2.2.4 Schedule Simulation

The schedule simulation tool provides timelines and manpower analysis for a given set of work instructions. Primary inputs are provided by the factory simulation and virtual assembly planning tools with results used for cost and risk estimation. Figure 2-5 shows the top-level interfaces for a typical schedule simulation tool.

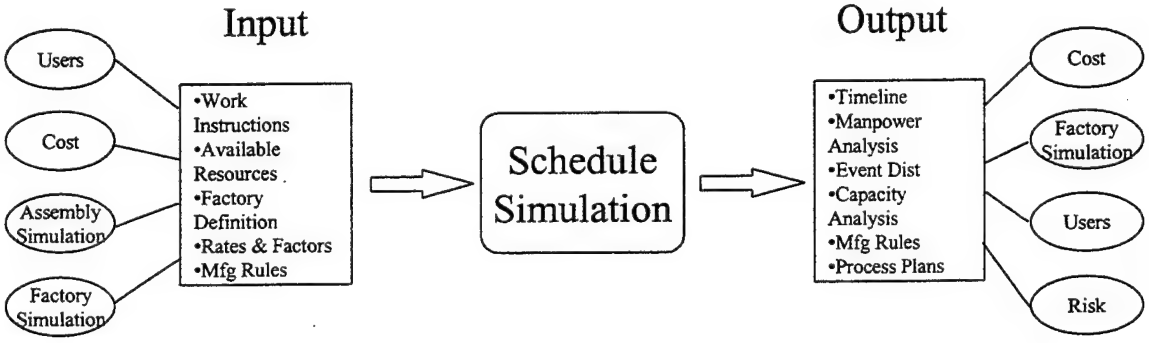

Figure 2-5 Schedule Simulation Interfaces

2.2.5 Cost Modeling

The cost modeling tool provides an estimate of the cost and producibility of a part containing a given set of features. The CAD tool provides the primary inputs for the cost model. This information, along with cost estimation models developed by users, provides the cost data, one of the primary drivers in assessment of a design alternative. Figure 2-6 shows the top-level interfaces for a typical cost modeling tool.

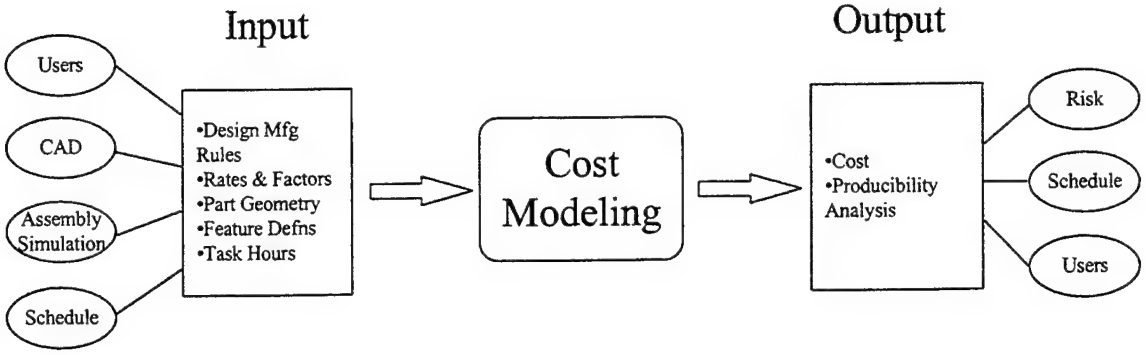

Figure 2-6 Cost Modeling Tool Interfaces

2.2.6 Risk Analysis

The risk analysis tool provides confidence profiles and uncertainty analysis for achieving a given set of parameters within a part or design study. Product definition, including tolerance and variability limits, are two primary inputs used in the risk estimation. Outputs from the risk analysis are used in the overall assessment of the design alternative. Figure 2-7 shows the top-level interfaces for a typical risk analysis tool.

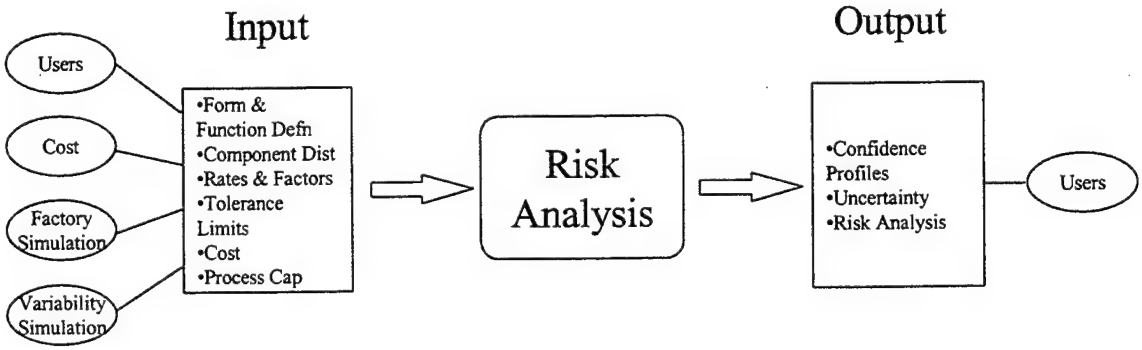

Figure 2-7 Risk Analysis Tool Interfaces

2.2.7 Assembly Variability Simulation

The assembly variability simulation tool uses CAD data, including features and tolerances, to make variability estimates for component and assembly distributions. This tool is tightly linked with the CAD tool and provides information to cost, schedule, and risk analysis tools. Figure 2-8 shows the top-level interfaces for a typical assembly variability simulation tool.

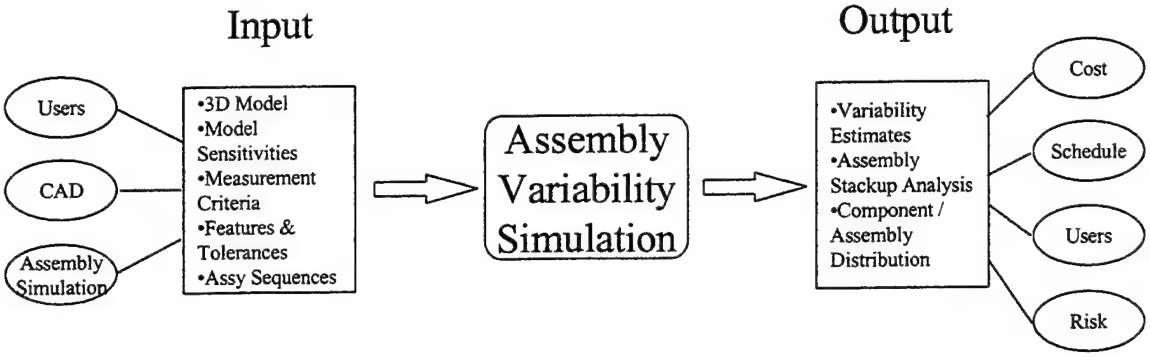

Figure 2-8 Assembly Variability Simulation Interfaces

Appendix A contains descriptions of the specific vendor tools that are being integrated into the SAVE environment. Functional concepts and flow diagrams are provided to acquaint the reader of the range of capabilities that are represented within these tools. These particular tools are being integrated to demonstrate the SAVE concept; however, the SAVE architecture and tool integration approach is applicable to any vendor tool that fits within the specified classes.

2.3 SAVE Data Model (SDM)

The SDM is the core component of the SAVE Architecture. The purpose of the SDM is to describe the data that need to be shared among manufacturing simulation software tools in order to determine the relative cost, schedule and risk impacts of a series of design studies. The model was developed using inputs from a number of sources:

- SAVE Tool Input/Output Specification Release 1.0
- Simulation Software Vendors
- Simulation Software Users
- Manufacturing Engineers
- Design Engineers
- The input from this variety of sources constitutes both a top down and a bottoms up definition of the data and data relationships common among the SAVE tool classes. The intent is not to represent all data for all simulations but to represent shared data for the classes of simulations within the SAVE environment.

In addition to common data, the model contains data that is necessary to assess the outcome of a series of design studies as well as model management data that provides access and organization to the model.

The SAVE model starts with a design study for a specific manufacturing program. The various alternatives associated with the particular design study are modeled and run through a series of simulations. At the heart of the model is the process plan that identifies the operations and resources necessary to manufacture or assemble a part/assembly. The plan and its associated part (or assembly) are assessed based on cost, schedule and risk at several levels of detail. Simulation results for these measures are compared and an alternative is selected as the preferred option for the design study. The model also allows for the situation where an assessment is desired for a single alternative. Figure 2-9 provides a top-level view of the SAVE data model.

This top-level model shows only the major interfaces (or classes) and their relationships with each other. The model contains five general categories of information, each representing information necessary within the VM environment. Model management data provide organization to the information stored in the SDM. One component of the model management data is the Simulation Request. This interface supplies the simulation tools with the necessary pointers into the SDM. The core process data include process plans and operations within those process plans. This is the primary set of data shared by the simulation tools. The resource data include tools, personnel, and work scheduling information that is applied to the operations in the process plan. Product data provide information about the part or assembly associated with the process plan along with pointers to the associated geometric or CAD data. Results data provide key decision making information about the relative cost, schedule, and risk of the alternatives under consideration. These components provide a robust mechanism for data sharing in the VM environment.

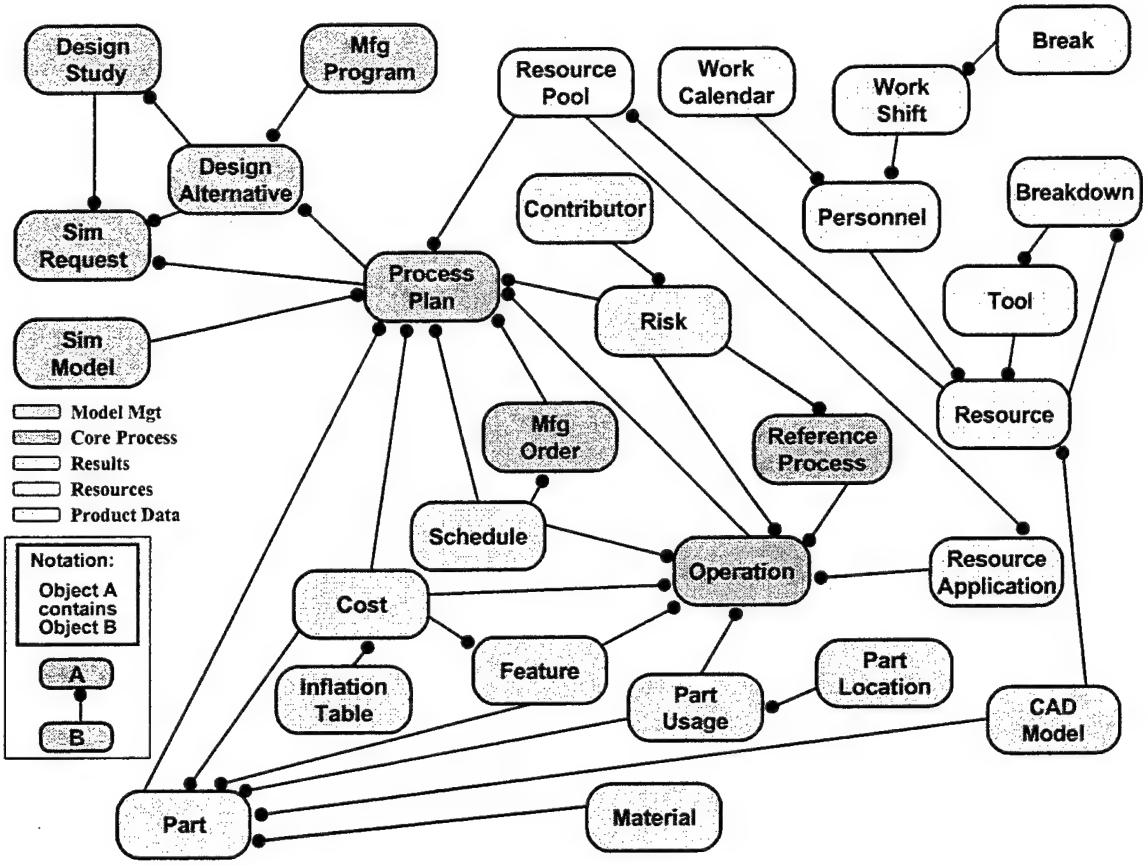

Figure 2-9 Top Level SAVE Data Model

2.3.1 Details of the SAVE Data Model

The SAVE Data Model is formally specified in CORBA Interface Definition Language (IDL). The IDL for the entire SDM is provided in Appendix E of this report. A data dictionary for the SDM is provided in the associated SAVE User Manual Report.

The classes of tools represented in the SAVE environment will interface with this model to exchange common information. Tools do not have to interface with all elements of the model; however there are certain items that are useful for all tools in data access and control. The simulation request interface provides a mechanism for a tool or a user to access information about the design studies, design study alternatives, and process plans that are currently stored in the database. This interface alleviates the need to traverse through the model to obtain information about these items and eliminates the cumbersome task of implementing back pointers throughout the model.

A typical manufacturing simulation tool will start with the process plan as an entry point and link to other relevant information from that interface. The top level interfaces, including design study, design study alternative, and manufacturing program, are more meaningful to the user in interpreting and managing the available information. The user, not one of the simulation tools, will typically populate these items using the Query

Manager. Most other data elements in the model will be used and/or populated by the simulation tools available as part of the SAVE environment.

2.3.2 Configuration Management in the SAVE Model

The nature of complex product design is inherently iterative and SAVE has been designed to manage the multi-version nature of design simulation data. As a design tool, SAVE-generated data are expected to be released (likely controlled by a PDM) to production and transferred to downstream systems. SAVE provides configuration management of data while it is in work, provides for data storage by a PDM, and allows results to be extracted to downstream systems if the data are not already stored there during development.

The philosophy behind SAVE data management is to provide flexible control that can be tailored by a design team. There is strong capability to create alternative approaches at several levels, lock each alternative as its study is completed, and to denote the selected alternative when the overall design study is concluded. Alternative approaches may be totally constructed from the ground up, or may be copied from an existing alternative when that is more efficient. Low level versioning of individual data attributes is provided to minimize the need to create a new version when only small changes are made. These versioned variables maintain a history of their values, and information on when and how they were changed. All data objects keep track of their references in other objects, and most user-initiated "deletions" are simply removal of a reference and a decrement in the reference count. When references to an object reduce to zero, the object is actually deleted from the database. Objects in SAVE libraries can only be deleted when all other references have been removed. See the SAVE Data Model Usage Guide section of this report for a complete description of the capabilities discussed above.

SAVE developers and users must develop an understanding of SAVE's Data Model and the data configuration management capabilities it provides. This understanding will allow a team to quickly identify the paths to be included in a design study and the best representation of the data within SAVE.

The elements of SAVE configuration management include:

- Status Flags - Included with several key data elements - lower level data are controlled automatically
- Alternatives - Supported for Design Studies and Process Plans
- Copy Command - Intelligent copy of Process Plan to start alternatives
- Remove/Delete - Tracks references to data objects by other objects
- Versioned Variables - Minimizes need to create alternatives
- Back End Data Storage - Data management of physical storage system

Each of these elements and their use are discussed below.

2.3.2.1 Status Flags

A Status Flag data attribute is included in the Design Study, Design Alternative, and Process Plan data objects within the SAVE Data Model. These flags can take on values of:

1. Working
2. Review
3. Released

Data can be modified in these objects only when the Status Flag is set to Working. Both the Review and Released status lock the data object from change. The Review status denotes that the data are in the review and approval cycle while the Released status denotes that the approval cycle has been completed. Users with appropriate authority can change the value of flag from a higher level to a lower level if it becomes necessary. This is in keeping with the fact that SAVE is a design study tool and when a study is complete, the data from the selected alternative is expected to be formally released through a PDM and passed to a downstream system.

While essentially all data within SAVE have a status flag, only the objects listed above expose their status flags to users for control. The status flags on lower level data objects are controlled by the objects in which they are referenced. For example, When a Process Plan status is set to "Released" all of its associated Operations are also locked. Likewise, the Cost data object referenced by an Operation is also locked. Some objects that are used in multiple Process Plans (for example Reference Processes) are not locked by a given Process Plan. Many Library objects are created specifically to be used repeatedly and should not be modified after their initial creation. If a modification is needed, a separate copy of the library object should be created and clearly noted in its Name and Description attribute fields.

2.3.2.2 Alternatives

The SAVE Data Model provides two levels of alternatives to be identified and included in a Design Study.

Multiple Design Alternatives in a Design Study

Multiple Process Plans in a Design Alternative

A Design Study may involve only a single approach to be studied and its cost, schedule, and risk assessed. But in many cases a team will identify two or more alternative approaches to a design problem and SAVE is an ideal tool to select the best alternative. Each of these first-level alternatives should be designated as a separate Design Alternative referenced by the Design Study.

A second level of alternatives is provided by the capability for a Design Alternative to maintain a list of possible Process Plans. This capability is included to allow a design team to consider one or more subtle variations on the basic Design Alternative. Use of these alternatives is not rigidly forced by the SAVE system, and it is up to the design team to plan their usage. Creating a Design Alternative is a very simple task and is performed by interactive access to the Data Model. Each alternative Process Plan can be created from scratch, or an alternative can be created by copying an existing Process Plan and making the desired changes as simulations are executed. This copy process will typically be performed by one of the design team members using the interactive Data Model access tool, currently the SAVE Query Manager.

In deciding what alternatives to create and use, the team must remember that there is no facility for merging two divergent plans, even if they were created as copies of the same plan. This is no different than having a team of people create a single word processing document. Some manual version coordination is required. Two alternatives may be created and studied, but ultimately one will have to be chosen as the final version.

Users should also understand and use the nested process plan capability within SAVE as it impacts alternatives. One Operation within SAVE can, in fact, be a sub-Process Plan. This nesting capability was originally included to support a high-level/low-level abstraction within Process Plans allowing some tools to simulate a higher level view while other tools simulate at a lower level. Plans can be nested to any depth, but typically only one to three levels are needed. Using this nesting feature, sub-process plans can be individually modified and a high-level process plan can be constructed to combine the best sub-plans. This is analogous to a team working on chapters of a report rather than working on multiple copies of the full report. A little creative thinking allows a very flexible capability.

2.3.2.3 Copy Command

An ability to easily copy some objects is provided to simplify the creation of alternatives. All data objects that are contained in a SAVE Library can be copied.

The copy capability is intelligent in that copying a data object automatically makes copies of all appropriate included objects. For example, copying a Process Plan automatically makes a set of Operations that are copies of the original set, and as one of these Operations is copied, its CostInfo, RiskInfo, and ScheduleInfo objects are also copied and associated with the new Operation.

The Copy command creates a new data object that is free to be modified without changing the original source data object. New copies of all included objects are made with the exception of data objects that are Library reference material **(Note: All Libraries are considered reference material except Design Study, Process Plan, and Sim Request)**. For example, when a Part is copied, the Material object from the original is referenced in the copied object. If a change is needed in the new Part, a new Material is copied, modified and used in the new Part. New objects created by Copy each have their own DataTime and copy all primitive data attributes. If an object is copied and it is

in a library or sequence that requires unique names, then the Copy command will automatically perturb the name (append a letter or number) to force uniqueness. Users may want to use the Query Manager to review and modify these names. The full hierarchy of copied objects is shown below:

- Break - No lower level copies
- CAD Model - No lower level copies
- Design Study

  Alternative (Sequence)

    Design Alternatives in sequence

      ProcessPlans (Sequence)
- Inflation Table - No lower level copies
- Material

  Unit Cost

    All Versioned Float objects

  Characteristics (Sequence)

    Characteristics in sequence

      Numeric Value
- Manufacturing Program - No lower level copies
- Part

  Cost

    All Versioned Floats

  Associated Parts (Sequence)

  Features (Sequence)

    Features in sequence

      Cost

        All Versioned Floats

      Characteristics (Sequence)

        Characteristics in sequence

          Numeric Value
- Personnel - No lower level copies
- Process Plan

  Cost

    All Versioned Floats

Schedule

All Versioned Floats and Versioned Strings

Risk (Sequence)

All Versioned Floats

Contributors (Sequence)

Contributors in sequence

MfgOrder

Characteristics (Sequence)

Characteristics in sequence

Numeric Value

SimModel (Sequence)

Sim Models in sequence

Tool Pool (Sequence)

Tool-type Resource Pools in sequence

Personnel Pool (Sequence)

Personnel-type Resource Pools in sequence

Operations (Sequence)

Operations in sequence

ProcessPlan - This is a non-reference Library object and must be copied. See Process Plan above.

Cost

Schedule

Risk (Sequence)

Precedents (Sequence)

New Operations are NOT copied here, but this sequence must reference appropriate copied objects

Characteristics (Sequence)

Characteristics in sequence

Numeric Value

PersonResApplic (Sequence)

Resource Applications in sequence

Pool

ToolResApplic (Sequence)

Resource Applications in sequence

Pool

Parts (Sequence)

Features (Sequence)

Features in sequence

Cost

All Versioned Floats

Characteristics (Sequence)

Characteristics in sequence

Numeric Value

- Reference Process

  Risk (Sequence)

  All Versioned Floats

  Contributors (Sequence)

  Contributors in sequence

  Characteristics (Sequence)

  Characteristics in sequence

  Numeric Value

  OpChar (sequence)

  Characteristics in sequence

  Numeric Value

- Simulation Request

  Design Alternative

  Process Plans (Sequence)

- Tool

  Characteristics (Sequence)

  Characteristics in sequence

  Numeric Value

  Breakdowns (Sequence)

  Breakdowns in sequence

  RepairResource (Sequence)

- Work Calendar - No lower level copies

- Work Shift

  BreakList (Sequence)

2.3.2.4 Remove / Delete

Each data object within the SAVE system keeps track of the number of times that it has been referenced by other data objects. In an object-oriented system such as SAVE, when an object is not referenced by another object, access to it is totally lost and it can (and should) be deleted. Data objects that need flexible long-term access, independent of other use within SAVE, are maintained in Libraries (just a special data object themselves).

Data objects track their own usage, totally freeing the user of this burden. Users are free to create objects and add or remove them from being referenced by other data objects. Therefore, all user "deletion" actions are actually reference removals. When non-library objects have their last reference removed, they are automatically deleted. A user can remove all references to a Library object, but the object remains because it is in the library. A user may remove a data object from the library, and if no other references exist, the object is deleted. Conversely, a user may remove a data object from the library (for example, remove a Reference Process that has been superseded) and it will not be visible there, but it will still exist if it is referenced in one or more Operation. As a point of note, deletion is actually performed the next time the server is shut down.

The SAVE data object removal / deletion scheme assures that no necessary data are ever lost and provides a scheme that is logical and simple for users to follow.

2.3.2.5 Versioned Variables

By the very nature of SAVE being a system for design studies, its data will be continually added to and modified. To maintain a record of this evolution would be prohibitive if new versions of the higher-level data objects were needed every time one low-level value changed. For instance, it would not be practical to have a new version of a Process Plan simply to track the fact that one cost element of one Operation had been updated.

SAVE has implemented a novel approach to this potential problem. Many of the rapidly changing low-level values are stored in Versioned Variables (Floats and Strings). These Versioned Variables are data objects themselves and they maintain the history of all values that a variable has held. Each value is stored along with a date-time stamp and a record of the tool that generated the value. By default, only the most current value is returned on simple queries, but any previous value (by date) or the entire history can be accessed if needed. If needed, the status of the Process Plan on a given date and time can be determined.

2.3.2.6 Back End Data Storage

The SAVE Data Model has been implemented to present a consistent view of stored data to SAVE-compliant client software while allowing the actual data storage to be distributed to multiple physical back end data stores. Most organizations that install SAVE will already have manufacturing process data stored in some electronic systems. SAVE was designed to minimize the requirement to replicate that data with its associated configuration issues. The SAVE Data Model server can read and write data to the existing database and still provide the standardized access to these data to client software tools.

The way that a SAVE server provides this distributed storage capability will likely vary among different commercial servers. During implementation users will identify which data objects and attributes they want to have physically stored in existing systems. This information will be fed to the server as data, and should require little or no recoding of the server software.

In many installations today, one of these back end data stores will be a Product Data Management (PDM) system. With minimal tailoring, a server can be extended to make calls to the PDM Application Protocol Interface (API) or CORBA interface. In this way the key data accessed from a SAVE server can be managed consistently and as formally as other product design data.

2.3.3 Units in the SAVE Data Model

All attributes explicitly defined in the SAVE data model have clearly defined units. These units are specified via either a separate units attribute within the object or a single unit requirement defined in the name and definition of the attribute. In some cases, objects have defined characteristics, which are name/value pairs, and allow for expansion of the attributes of an element. These characteristics vary in type; therefore, units cannot be defined in advance.

The numerical characteristic values are stored with the defined type, value with units. This format allows for definition of a value with its associated units. The value with units type uses data defined in a standard units object containing definitions of the standard set of units needed in the model as well as conversion factors to other commonly used units. This standard units object is defined by the user. Some examples of these standard units are shown in Table 2-1.

Table 2-1 Samples of Value With Units Standardization

Type	Unit
length	foot
area	square feet
cost	dollar
weight	pound

Unit	Conversion Factor	Unit
yard	3.0	foot
inch	0.83333	foot

The units string within the msmValueWithUnits object provides the capability for defining compound units (e.g., dollars/pound, pound/square feet). As these values are stored in the units string, a validation operation is performed to verify that the units are defined in the standard units object. In addition, any necessary conversions may be made at that time. If the units are not valid, an error is raised.

The use of the value with units object provides a flexible mechanism for defining the units of characteristics and for their interpretation by the tool interfaces. When a tool defines and populates a characteristic object, it stores any numerical value as a float with a text string. The text string defines the units for the float value. Any tool using the characteristic will read the value with units and may use the defined conversion values along with the conversion method to product the value in the desired units.

2.3.4 Libraries in the SAVE Data Model

The SAVE data model employs the concept of libraries to provide user access to certain groups of objects. The library object is simply a list of all SAVE instances of a particular object type with pointers to their persistent location in the data store. There are find-by-name and find-by-index methods within the ObjectSeq object that provide access to the objects in a library. Library objects are populated automatically as new SAVE objects that are part of the library are created. Currently, the SAVE model provides for the following libraries:

- Break
- CAD Model
- Design Alternative
- Design Study
- Inflation Table
- Material
- Manufacturing Program
- Part
- Personnel
- Process Plan
- Reference Process
- Simulation Request

- Tool
- Work Calendar
- Work Shift

Some library objects will be populated manually, using the Query Manager, prior to execution of a design study. These objects contain information that may be used many times. For example, the reference process object contains standard information about a process that may be accessed by an operation using that process. Automated loading tools, written as clients to the SDM server application, or simulation tools that already contain the desired information may also populate library objects.

2.3.5 Resources in the SAVE Data Model

In order to accommodate complex resource utilization and tracking, the SDM contains a rather extensive resource model. The resource model includes the following objects: resource, tool, personnel, resource pool, and resource application.

Personnel and tools are generic library objects that define the types of resources that are available to any process. Typically, at the trade study level, it is necessary to identify the resources required, not necessarily individual work assignments.

A process plan can have many different types of resources available, each of which is described in a resource pool. The pool for personnel or tools identifies the quantity of a particular skillset or type and allows tracking for over or under utilization. Any operation in a process plan can draw from one or more of the resource pools associated with its process plan. Each use of a resource pool by an operation is a resource application. The resource application provides information about the location and quantity of the resource with respect to the needs of the process and the tool.

2.3.6 Control of Duplicate Names Used in Object Sequences

The majority of SAVE data objects are used in lists (Object Sequences), either referenced by other objects, or in Libraries (which are simply specialized sequences). Data objects are inherently unique and it is their unique ID that is listed in a sequence. User interaction and understanding of data in the SDM requires that some data objects have unique names. This is enforced by the SDM Server as it populates sequences. The set of uniquely named objects is controlled by the end users at a given site by identifying these object types in a file which is read by the Server as it is started. A summary of SDM objects, whether they are Base or Named objects, Library objects, used in sequences, and uniquely named (for SAVE development implementations) is shown in Table 2-2 below.

Table 2-2 SDM Data Object Characteristics

OBJECT	BASE/ NAMED	LIBRARY	SEQUENCED	UNIQUE NAME
Base Object	-	-	-	-
Break	N	L	S	U

OBJECT	BASE/ NAMED	LIBRARY	SEQUENCED	UNIQUE NAME
Breakdown	N		S	U
CAD Model	N	L	S	U
Characteristic	N		S	U
Contributor	N		S	U
CostInfo	B			
DbAccess	-			
Date-Time	B			
DesignAlternative	N	L	S	U
DesignStudy	N	L	S	U
Feature	N		S	N
InflationTable	N	L	S	U
Library	N			
Material	N	L	S	U
ManufacturingOrder	N		S	U
ManufacturingProgram	N	L	S	U
NamedObject	B			
ObjectSequence	B			
Operation	N		S	U
Part	N	L	S	U
PartLocation	N		S	U
PartUsage	N		S	U
Personnel	N	L	S	U
ProcessPlan	N	L	S	U
ReferenceProcess	N	L	S	U
Resource	N			
ResourceApplication	N		S	U
ResourcePool	N		S	U
RiskInfo	N		S	N
ScheduleInfo	B			
SimulationModel	N		S	N
SimulationRequest	N	L	S	U
Tool	N	L	S	U
ValueWithUnits	B			
VersionedFloat	B			
VersionedString	B			
WorkCalendar	N	L	S	U
WorkShift	N	L	S	U

2.4 SAVE Work Flow Manager

The SAVE architecture contains a Work Flow Manager (WFM) that provides graphical process modeling and execution. The SAVE team developed this software with strong

influence from the DARPA Simulation Based Design (SBD) Program. It is implemented in JAVA to provide full platform independence. As depicted in Figure 2-10, the WFM software defines dependency relationships among the components of the process with decomposition down to the activity level. When the WFM executes, it has the capability to send messages to both users and tools, monitor progress and status, and provide graphical feedback to the user. The use of this tool is not discussed in detail within this specification; however, the interface between the SAVE simulation tools and the WFM is addressed. The IDL for the simulation tool interface to the WFM is provided in Appendix E of the report. The Work Flow Manger User's Guide is contained in the associated User's Manual Report and provides detailed instructions on the use of the tool.

Many of the manufacturing simulation tools within SAVE are interactive and do not run in a batch mode. The Work Flow Manager recognizes this fact and provides for an email to be sent to the correct user when an activity is prepared to run. When the user is ready, and has started the simulation tool, she uses the WFM to Resume the paused tool.

2.4.1 Definition of Terms

In order for a developer to fully understand the discussion of the Work Flow Manager and its interface with simulation tools, it is necessary to define the terminology used in this discussion.

The WFM is designed to allow users to create dependencies and nesting at several levels within the work flow. The process is the top-level node for the work flow model. It may contain other processes and tasks, but may not contain an activity. A process organizes the overall work flow

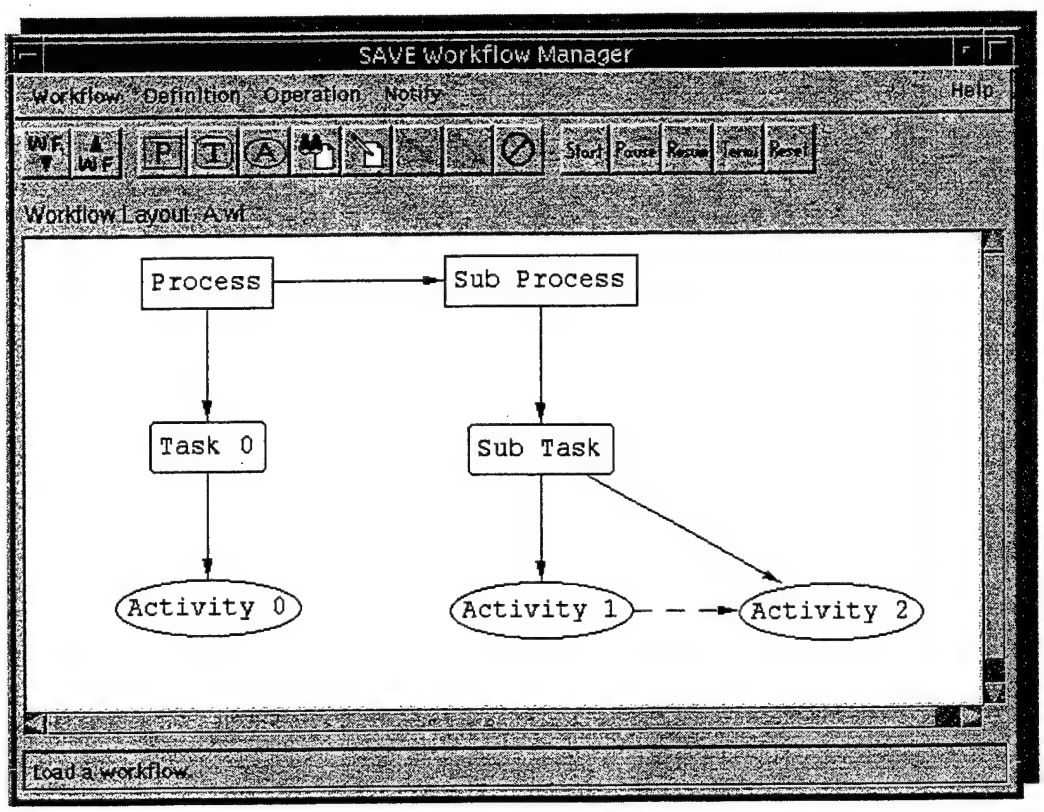

Figure 2-10 SAVE Work Flow Manager

and may have only one predecessor node. It does, however, support multiple successor nodes. A task provides a container for the activity nodes. A task is the intermediate level of organization within the work flow and can have only one predecessor node, which must be a process. Tasks support multiple activities as successor nodes. An activity is the lowest level node in the work flow. Activities are the only entity within a work flow that may interact with the interface to the simulation tool. An activity may have multiple activities as predecessor or successor nodes. An activity may have only one task predecessor. When an activity completes, its successor activities start. When all of the activities are complete, the encapsulating task node completes. This feedback continues until the entire process is complete.

A simulator is the interface between the WFM and the simulation tool. It is a CORBA server that sits on top of the simulation tool and controls its operation based on information supplied by the Work Flow Manager CORBA client's activities.

A listener is a transient object that is created by the WFM and registered with a simulator. The simulator keeps the WFM informed of its progress and status by sending messages to the listener.

2.4.2 WFM Messages and States

Once the user establishes the work flow, the WFM executes the sequence of instructions by sending messages to the simulation tools via CORBA. The tool wrapper, called a

simulator in this case, is responsible for interpreting the WFM messages and delivering status information. There are five messages, shown below, that the WFM may use in communicating with a simulator. The simulator, in turn, responds to these messages with one of nine possible states. Figure 2-11 shows a typical message/state interaction between the WFM and a simulator.

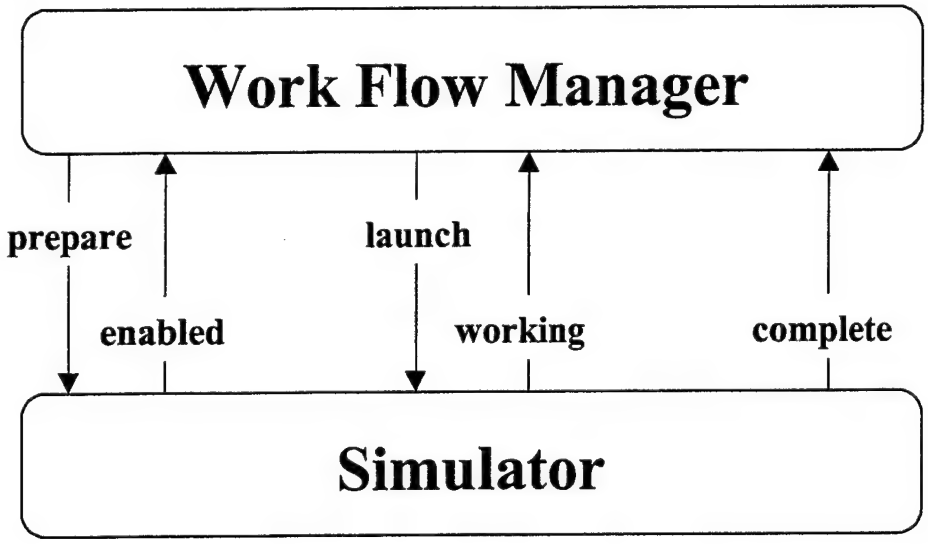

Figure 2-11 Typical Interaction Between the WFM and a Simulator

The Prepare operation prompts the simulator to perform any necessary initialization steps and prepare for execution. In this step, the WFM provides the simulation request that will be used in the simulation. The simulation request contains the design study, design study alternative, and process plan from the SDM that should be used by the simulation code in its execution. It also contains optional information relating to launch instructions, input files, and input/output options. The simulator sends state information back to the WFM after preparing for execution. Possible state responses include undefined, initialized, or enabled. Once the tool has successfully accessed the simulation request and is prepared, the enabled state is returned. The WFM changes the icon to the color blue to signal its user that the simulator is enabled.

The Launch operation sends a message to the tool to start its execution. Upon launch, the simulator sends the working state to the WFM and its icon changes to the color green. The simulator will continue in the working state during the execution of the manufacturing simulation. During this time, the user may perform numerous iterations with the tool and will interact accordingly with the SDM. If there are problems during the simulation, the simulator will update its state to indicate its current condition. Possible states include terminated or faulty. Once the simulation is complete and the appropriate results have been stored in the SDM, the simulator updates its state to completed. At this point, the icon changes to gray to indicate that it is complete and the WFM is ready for the next step.

The WFM may also instruct to the simulator to terminate, pause, or resume. The corresponding states for these operations are terminated, paused, or resumed. A

terminated simulator will be unable to complete its simulation and loss of data may occur. This state is indicated with a red icon. When a pause instruction is sent, the simulator will halt tool execution at a reasonable checkpoint so that it may be resumed without loss of data. A yellow icon represents a paused state.

It is not necessary for a simulator to support all operations and states available from the WFM. The developer should implement those that make sense for the specific simulation tool being integrated, and the WFM will poll the simulator to verify which operations are applicable.

2.4.3 Listener Objects

Simulators communicate events to the WFM using listeners. A listener is an object created by the WFM and registered with a simulator. Once a listener is registered, it is the responsibility of the simulator to notify the listener of any state changes so that they may be communicated back to the WFM. The simulator accomplishes this task by invoking methods on the listeners. In addition, the simulator must supply its CORBA name and ID as part of each method call in order to identify itself to the listener and, in turn the WFM.

2.5 SAVE Query Manager

In order to provide visibility into the SAVE data, the team developed a Query Manager (QM) application. This JAVA application provides the capability to browse, create, modify, and delete objects in the SDM directly. Figure 2-12 shows the screen layout for the QM. The left-hand side provides a tree structure of the library objects that exist in the model, whereas, the right-hand side displays attribute data for a specific object within the tree.

The SAVE QM does not interface directly with either the simulation tools or the WFM. Its sole purpose is to provide access to information in the SDM through a mechanism other than the simulation tools within the SAVE toolsuite. The Query Manager User's Guide, included in Appendix M of the Software User's Manual, provides detailed instructions for use of the QM application.

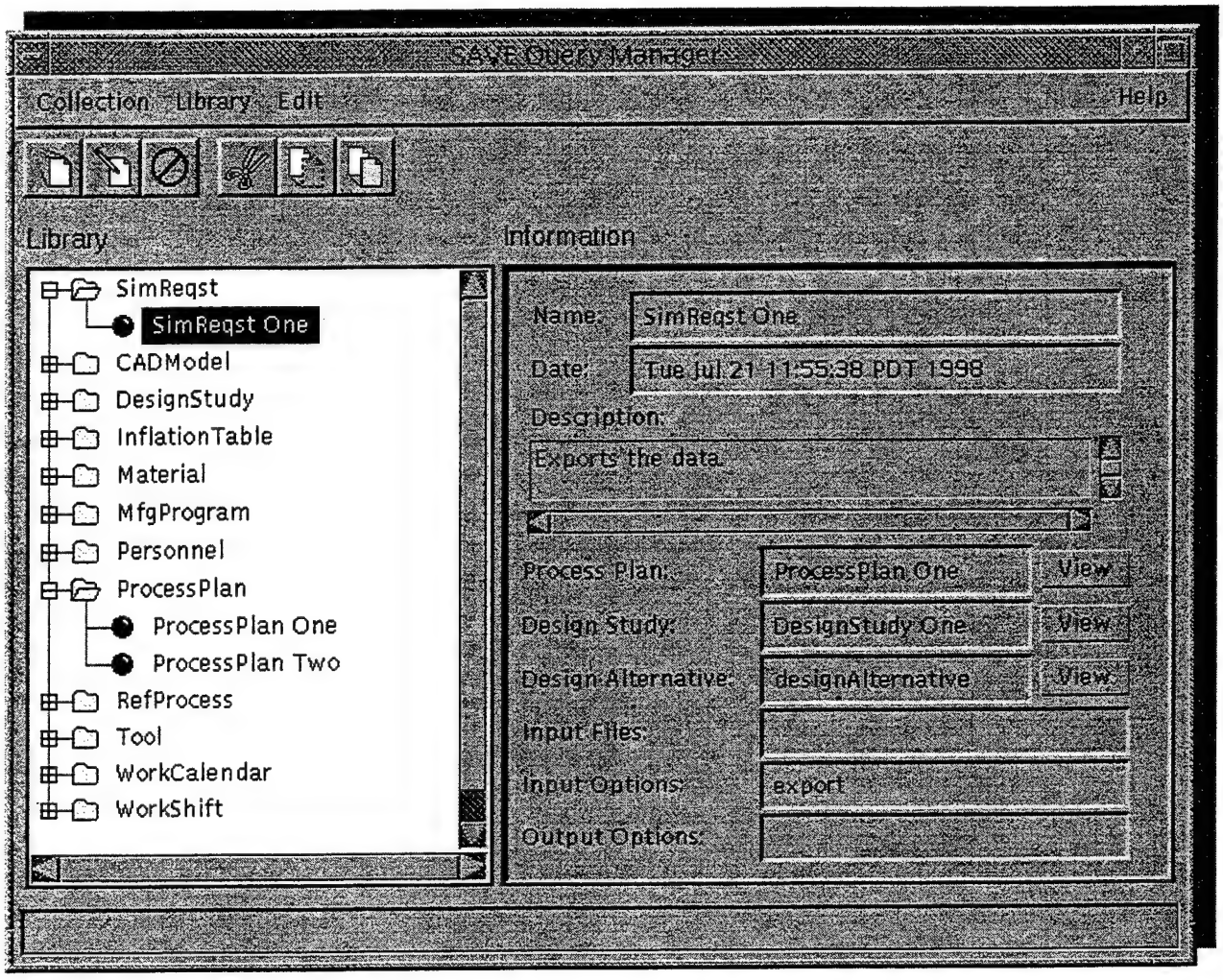

Figure 2-12 SAVE Query Manager

2.6 Use of CORBA for Integration

A primary objective of SAVE tool integration is to provide a mechanism for data exchange among manufacturing simulation tools that is independent of both data location and storage mechanism. Several options for data exchange were investigated with this objective in mind. One option included storing the shared data in a database (either relational or object oriented) and writing tool interfaces directly to the database. This approach proved undesirable because of its reliance on a specific database product. The product dependence would cause one of two situations: users would be forced to store their information in the specific database product chosen for SAVE or tool vendors would have to reprogram their interfaces for every database product desired by a user. The second option reaps the advantages of database storage without the burdens by adding an abstraction layer between the tool vendor applications and the database. The Object Management Group has developed the Common Object Request Broker

(CORBA) standard for abstracting this information, thus, making it independent of application and platform.

In order to provide seamless communication among the components of SAVE, CORBA was selected as an abstraction layer between both the data storage mechanism and the Work Flow Manager and the tool interfaces. CORBA is an object-oriented system for integrating distributed computing systems. Development of the CORBA standard is managed by the Object Management Group - a not for profit company with over 800 members worldwide. Specifically, CORBA provides middleware functionality for integrating distributed systems without regard for hardware platform, operating system, network protocols, or application language.

The CORBA standard contains two primary components: Interface Definition Language (IDL) and Internet Inter-ORB Protocol (IIOP). IDL provides a language independent object specification whose implementation is hidden behind the interface. The IDL serves as the contract between the SAVE client and server developers. Adherence to the contract insures seamless integration. IIOP is the communication protocol that provides transparent communication among distributed objects. Using IIOP, the SAVE data may be located on any platform or operating system and still be accessible by both the client and the server. Figure 2-13 provides a simple view of a typical CORBA client server application.

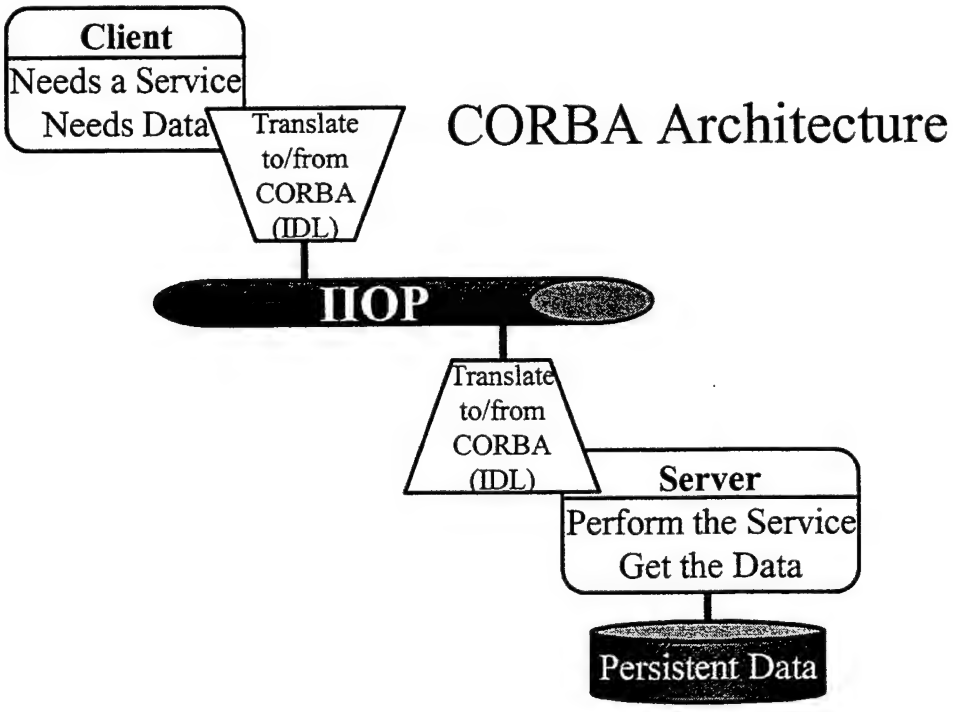

Figure 2-13 CORBA Client Server Application

The CORBA architecture provides benefits for the SAVE program. First, the tool vendors develop a single interface for data exchange with the other tools and for tool execution via the WFM. This interface does not change as new tools are added to the

suite because every tool interface is developed to adhere to the IDL. This concept eliminates the need for and the expense of developing point to point interfaces among the various vendor tools. In addition, CORBA provides implementation flexibility. The use of IIOP allows the data storage mechanism to be defined by the deployment sites, not the SAVE development team. The data server software will "collect" the data represented in the SDM from various data storage solutions (including relational databases, object-oriented databases, and product data management systems) and provide seamless access to the vendor tools.

The strategy for effective use of CORBA within the SAVE program involves creating IDL files that are based on the SAVE data model and the WFM functionality and distributing that IDL to tool vendors interested in developing an interface into the SAVE environment. The data model and both IDL files are provided in Appendix E of this document.

3.0 Integration and Development Process

The SAVE specification identifies two types of CORBA interfaces necessary to integrate a VM tool into the SAVE infrastructure. The first is a CORBA IDL definition of the interface that allows the manufacturing simulation tools access to information within the SAVE Data Model (SDM). This interface facilitates data sharing among the tools. SAVE-compliant simulation tools become clients to the SDM server. Effort to interface with the object-oriented data model varies with the amount of input/output required but is estimated to require between 200 and 300 person hours.

The second type of interface provides communication between the Work Flow Manager (WFM) and the manufacturing simulation tools. Using this interface, the tools can accept inputs and commands from the WFM as the process executes. In this case the simulation tool, or its wrapper, becomes a CORBA server to the WFM client. Approximately 40 person hours are required to create this interface with the infrastructure.

Both of these interfaces are developed only once for each tool that is integrated into the SAVE environment. As long as the SDM definition and its CORBA IDL do not change, additional tools can be added or physical data storage locations can be changed, and a SAVE-compliant simulation tool will continue to operate without modification.

During the development of any system that has several software layers and tight integration among a suite of tools, it is important to understand the steps in the process and the responsibilities of the parties involved in the development.

This document and the associated CDROM contain all the information that is needed to develop SAVE-compliant tools including servers, work flow managers, or simulation tool wrappers. Primary specification data are the CORBA IDL files for the SDM and the Work Flow Manager. The following chapters contain background information and the specifics of the software that was developed under the SAVE contract to validate and demonstrate the SAVE integration approach. Sample code is included in Appendix D and on the CDROM to support rapid spin-up of developers who are new to CORBA.

3.1 Tool Interface Development – SDM

A brief description of the SAVE data model client operations will help the developer/implementer understand how the model will be accessed. The details discussed below will be largely transparent to the simulation code user, with the exception of infrequent delays when two simulation runs try to update the object model simultaneously. This access scenario is important, however, to the interface development team.

In general, any simulation tool that is SAVE-compliant will access the database in two separate transactions during its execution. Prior to simulation execution, a read transaction will be used to access the data needed as input to the simulation. This read transaction will be non-blocking, that is, one update transaction and other read transactions may occur simultaneously. The read transaction will end when all input data

has been read. When a simulation is complete, an update transaction will be initiated to place new data into the database. If another update transaction is in progress, this request will wait until the first is finished. It is estimated that this wait should be no longer than few minutes in length to achieve acceptable performance. Once an update transaction is initiated, new data will update appropriate objects in the model, and the transaction will be completed with a database commit.

The SAVE IDL includes a database access object that contains methods for database transaction control including commits or rollbacks as well as for object creation. This database object contains no data and is not stored in persistent storage. It is easily declared in the client code, making its methods available to the client. An object of this type will be created by the client to make the persistent data store active and to invoke transactions via the CORBA layer. The database access object allows for direct access to two types of objects: library and simulation requests.

In the SAVE concept of access, the clients will not have general query capability; however, entry into the data model is provided via the simulation request object. The SAVE work flow manager will launch a simulation code that has a CORBA interface into the SAVE environment and will pass a reference to a particular simulation request locator object in the data model. This locator will provide references to the required design study, design study alternative, and process plan for which the simulation run is to be made. These object pointers are sufficient to access all related data within the model. The only other access mechanism available via the SAVE data model is the library object.

Within the SAVE data model, most data variables are directly accessible for either read or update. A few variables must be updated through methods to assure consistency. The versioned variables in the cost, schedule, and risk interfaces are examples of these. Clients can access a list of versioned variables along with their value, source, and date/time. In addition to direct access to individual data variables, the SDM provides several mechanisms to access objects that are attributes of other objects. The sequence of objects may be retrieved at one time and made available to the client. In addition, a structure is available to provide the name and description along with the objects themselves.

Every object in the SDM inherits directly from either the msmBaseObject or the msmNamedObject class. Currently, the msmBaseObject does not contain attributes that are inherited by its child classes. This class was created to allow for future extensions if the need arises. The children of msmNamedObject inherit name, description, and datetime attributes.

Several guidelines were established in the SDM for object creation. When any client performs a create object operation, the constructor for that object will automatically create the base objects that are attributes of that object. These objects are defined as readonly attributes of their parent object and must be used by the client in lieu of creating these objects separately. Constructors will not automatically create named objects that are attributes of other objects. It is the responsibility of a client to check using the

CORBA IS_NIL() function to determine whether the object exists. The difference between named objects and base objects in the SDM provides the rationale for this delineation. Many named objects are part of libraries and may be used in more than one object attribute. For this reason, the named objects are not automatically created. Base objects, on the other hand, are typically complex sets of data that are a part of another object. Although structures are usually utilized to capture complex sets of data, objects are necessary in this case to allow for implementation of methods to operate on the data.

During update operations, the SAVE server will automatically assign date/time stamps to the named objects, so the client will not be responsible for that function. When updates are complete, the client may give a commit command to signal a successful update or a rollback command to signal a problem. Once a commit or rollback is executed, the transaction ends.

3.2 Tool Interface Development – WFM

A brief description of the WFM activity element will help the integrator understand how the WFM will interact with the simulation tool.

The lowest level entity available to a user designing a work flow is an activity. The activity contains information about the simulation tool that will be executed. In developing a work flow, the user selects a simulator (manufacturing simulation tool wrapped with a SAVE-compliant work flow server) from a list of those available in the environment. The simulator, in this case, corresponds to a vendor simulation tool. The user also specifies the location of the simulator, the name of the Simulation Request data object that will be used by the tool, and the email address of a user if the tool is to be run in a interactive mode..

When an activity starts, the WFM binds to its assigned CORBA simulator object (the portion of a tool wrapper that implements a work flow server). In the case of interactive tools, the user receives an email, starts the simulation tool and its server, and accesses the WFM to indicate his availability, after which the WFM performs the bind. The WFM then creates a simulator listener and registers itself as a target object so that it can receive events from the listener. Next, the WFM adds the listener to its simulator server so that when the simulator generates an event, the WFM activity receives it via the listener. In this case, an event is any communication from the simulator. For example, a state update from *enabled* to *working* is an event. Once the listener is created and appropriately registered, the activity sends the initial command to the simulator.

A simulator has a single method for invoking operations. The method arguments consist of an enumerated simulator operation and a command string. Although there are only five simulator operations, as discussed in Section 2.4.2, there are an unlimited number of commands for any given operation. The developer of a particular simulator server defines these commands, which are specific to the simulation tool being integrated into the SAVE environment.

A simulator is initialized in several steps. The WFM invokes the getOperations() method to identify which of the five operations the simulator recognizes. If the prepare operation is among them, the WFM will obtain information about the initialization process and send the appropriate information to the simulator. If prepare is not supported, there is no initialization required and the WFM may move to the launch command. In general, there are two methods that support each operation. The getCommand() method queries the simulator for the commands that have been defined for that particular operation. For example, the launch operation may support an import and export command, depending on the purpose of the simulation run. Once all supported commands and operations are known, the WFM will execute the doOperation() method to tell the simulator to perform one of the five operations in context of the command provided. One of the status messages, discussed in Section 2.4.2, is passed back to the WFM in response to the simulator's execution of the command.

3.3 User Interaction and Responsibilities

Although the SAVE data model and its implementation provide a mechanism for sharing common data among simulation software tools that is transparent to the user, the users of the toolsuite will need some knowledge of the model and how they wish to apply it. In order to facilitate this process, the SAVE team has provided training material for users as well as a Query Manager for use by the user community.

The SAVE data model has been developed with great flexibility so that it may be applied to any manufacturing domain (e.g., aerospace, automotive); however, with this level of flexibility comes a cost. To fully utilize the model, users will need some knowledge of its capabilities. For example, the mechanism for naming operations within a process plan is broad in order to allow companies to use their own conventions. Before executing a simulation with the toolsuite, users will need to agree upon names that can be supported by the tools they are using. Users will also need to decide upon the order that the tools will be used and which data elements each tool will use and populate. These requirements are more fully described in the SAVE User Manual Report in the chapter on SAVE concept of operations.

To provide access to the SDM from an application other than the vendor tools, SAVE includes a Query Manager application. This application will give users the capability to browse, create, modify, and delete all parts of the data model. Users will be able to see all information associated with a particular design study or populate design study and design study alternative objects upon initiation of a new activity. In addition, the Query Manger provides a mechanism for access to and population of libraries.

3.4 Developer Responsibilities

The IDL defined in Appendix E of this specification is the contract between all developers of "SAVE compliant" software. Developers should become familiar with the IDL as well as the notes and guidelines provided in the embedded comments.

SAVE-compliant capability can be added to an existing simulation tool in two ways. First, a separate wrapper code can be written to access the SDM and convert data to/from the native data format of the tool. Second, SAVE-compliant data access methods can be integrated into the simulation tool directly. The choice is left up to the simulation tool developers. It should go without saying that ease of use should be a controlling factor. Wherever possible, input and output options should be provided to allow users more control over the role the tool plays in the SAVE environment.

The flexibility built into the SDM may create some challenges in the interface development activity. Developers are encouraged to use the flexibility to produce software that is versatile within the wide range of possible applications in manufacturing simulation. One example of this flexibility is the fact that SAVE has not rigidly forced naming for many attributes within the SDM. For example, it is up to the end users to establish conventions for naming Operations within a Process Plan. When a design team uses simulations and models developed for a particular case, consistent naming will be easy to obtain. When the team desires to use previously generated models, the naming may be inconsistent. SAVE software developers should be sensitive to this potential problem and should provide a means of mapping names used in the simulation tools to the names used in the SDM.

4.0 SAVE-Developed Software Description

This chapter describes the software that was developed to demonstrate and validate the SAVE approach to integration of manufacturing simulation tools. The software programs that will be described include:

- SAVE Data Model Server - This program implements the access to and persistent storage of the data that are shared among the simulation tools. The server documented here is written in C++ and uses ObjectStore from Object Design for persistent storage.

- Query Manager - This JAVA application/applet provides interactive access to data in the SAVE Data Model. It acts as a client to the server listed above.

- Work Flow Manager - The work flow manager is a JAVA application/applet that allows a design team to organize and automate the order in which simulation tools are executed during the design study.

- Microsoft Project Wrapper - MS Project was wrapped as an exercise by one of the SAVE beta test teams and is used to create a first cut version of a process plan with initial schedule and resources identified.

Six commercial manufacturing simulation tools were also wrapped as part of SAVE development, but will not be documented here. The SAVE-compliant wrappers for these tools are unique to the commercial tools and can be obtained from their suppliers:

- IGRIP/ERGO - Assembly simulation from Deneb Robotics

- QUEST - Factory simulation from Deneb Robotics

- CostAdvantage - Knowledge-based cost modeling from Cognition Corporation

- FactorAim - Factory/Schedule simulation from Pritsker Division of Symix Corp.

- VSA-3D - 3-D geometric dimensioning and tolerance analysis from Engineering Animation Incorporated

- ASURE - Risk analysis from Science Application International Corporation

4.1 SAVE Data Model Server

4.1.1 Programmer's Guide

4.1.1.1 Map to Code Organization

This first group of files relate to the CORBA Object Request Broker (ORB). SAVE development to date uses ORBIX and ORBIX WEB from IONA.

- save.IDL - This is where the SAVE objects are defined. The CORBA objects are defined as interfaces with methods and attributes as in object-oriented class structure. The CORBA methods and attributes are both mapped to input/output functions.

- save_i.hh - ORBIX compiler-generated header file. This file needs to be included in all client and server code. It contains the C++ mapping of the IDL objects.

- SaveC.cpp - ORBIX compiler-generated server file. This file needs to be included in all server files, along with save_i.hh.

- SaveS.cpp - ORBIX compiler-generated client file. This file needs to be included in all client files, along with save_i.hh.

- save_i.h - This is the header file for defining the C++ class structure, mapped from the IDL interfaces.

- save_i.cpp - This is the bulk of the server code. All C++ functionality is coded here. The bulk of this code is mapping CORBA objects into C++ objects and storing and reading them from an ObjectStore database (save.db).

- server.cpp - The server initialization file that checks for client communication and binds requesting clients to the SAVE server, and hence the database.

The following files are related to the ObjectStore Object-oriented Database from Object Design Inc.

- Schema.scm - Schema file for marking C++ objects to be stored in the ObjectStore database.

- Csave.hh - C++ file for outlining the database objects and their relationships.

- save.db - The name of the SAVE database where all objects are stored. This database will be created automatically if one does not exist or is used by the SAVE server if it does exist.

- save.adb - A compiler generated file used by ObjectStore to assist in database mapping.

The following are general-purpose files:

- duplicates.txt - A file read by the server to mark which IDL sequences allow duplicate objects.
- Makefile - Makefile for compiling the CORBA, ObjectStore and C++ code.
- constant.h - A file used to reference the length of character arrays in the ObjectStore database.

4.1.1.2 Required COTS Products to Modify/Run the SAVE Data Model Server

The SAVE server uses the following commercial software products in its implementation:

- ObjectStore v5.1
- ORBIX for WINNT/Win95 v2.3c
- Visual C++ v5.0 (other C++ compilers could be used with a different makefile)

4.1.1.3 Implementation Approach

This section describes some of the implementation approaches and decisions that were made during development of the SAVE Data Model Server.

ORBIX - There are two approaches CORBA can use to bind objects between a client a server, BOA and TIE. At the time the SAVE server development began, using ObjectStore required the TIE approach. ORBIX claims that "these (two approaches) do not differ greatly in their power, and it is frequently a matter of personal taste which one is preferred". SAVE development adopted the philosophy to make the server as portable and platform independent as possible, therefore the ORBIX implementation was kept relatively simple. Very generic attribute types were used and did not incorporate any ORBIX add-ons that might not be robust or efficient.

ObjectStore - In using ObjectStore functionality, the development team again tried to use simplified code to keep the server execution as efficient as possible. Although ObjectStore can store all C++ object types, only very basic types (float, char and numeric arrays, integers, and structures) were used. Of the ObjectStore specific functionality for database storage the team used os_list collections. These are ordered collections that are mapped almost directly to the IDL sequence objects.

Proxy Objects - Because of the overhead associated with network communication, ORBIX objects are very large in size. Each simple attribute type inherits from complex ORBIX objects that have built in functionality to support network client-server communication. This information, while meaningful to the CORBA protocol, is meaningless to SAVE. Therefore, to cut down on the database size and object complexity, the meaningful data from the CORBA objects was stripped and stored as

C++ data types. When retrieving these objects from the database, new CORBA objects are created and the meaningful data are stored in them.

Database Partitioning/Roots - The SAVE Data Model is very vertically structured. There is one top level object, DbAccess, that is the starting point for all SAVE database navigation. From DbAcess, navigation of the database involves getLibrary or getSimReqst functions. These are the only two roots into the database. Also, since most navigation involves drilling down through instantiated objects there is no database segmentation.

Database Locking/Transactions - Database locking is handled through C++ code, written to support some rules regarding transactions. A brief summary of the rules is as follows:

1) Only 1 Update (persistent) transaction is allowed at a time.

2) An Update transaction creates a lock on the database, prohibiting other attempts at Update transactions. An exception is thrown if an attempt is made.

3) Any number of Read transactions can occur at any time.

In the server code, the read and update transactions are both treated as ObjectStore write transactions. This is due to conflicts in ObjectStore trying to access the database with different types of transactions at the same time.

Persistent Server - The SAVE server is persistent, running continually waiting for client connections. Therefore, the server needs to be started before any clients attempt to access the SAVE data.

Miscellaneous - Code has been added to control the usage of duplicate names in the database. Duplicate names are not a problem for C++, ORBIX, or ObjectStore since all of these products manage objects by their object references. Unfortunately, some client programs reference objects by their names and cannot support duplicate names . Hence, a controlling duplicates.txt file is used to determine, at database startup, which if any object types will allow duplicate names.

4.1.1.4 Steps to Code Compilation

Before compiling :

1) Install ObjectStore 5.1

2) Install ORBIX 2.3c

3) Install the source files onto a Windows machine. For reference below we will assume this directory is called c:\temp\save

To Compile:

1) Open a DOS shell, navigate to the SAVE directory

2) Type 'nmake all'

To Compile the SAVE client only: (server compilation can be time consuming)

1) Open a DOS shell, navigate to the SAVE directory

2) Type 'nmake save_client.exe'

The included makefile includes the commands needed for compiling the Oribx IDL, the C++ client and server code and the embedded ObjectStore code. This makefile uses Visual C++ but a few changes could be made to use a more generic C++ compiler. Makefile options are:

- nmake: This will recompile all code that has changed since the last compile.
- nmake save_client.exe: This will compile the client executable, assuming changes were made to the source code.
- nmake clean: This will delete all executables created by the nmake. This is useful for a forced recompilation of all code.

4.1.1.5 Testing

To Install and run the SAVE server:

1) Start the ORBIX daemon, either from the Windows groups icons or by typing 'ORBIXd' from a DOS prompt.

2) Register the SAVE server with the daemon, from a DOS shell, type 'putit save c:\temp\save\server.exe' (assuming c:\temp\save is save dir).

3) Edit the SAVE server user rights, either from the ORBIX Server Manager GUI or from the DOS prompt by typing 'chmodit save i+all' and 'chmodit save l+all'.

4) Execute the SAVE server, by typing 'server' from a DOS shell navigated to the save directory.

To run the sample client:

1) Open a DOS window, navigate to the save dir, type 'client <machine>' where <machine> is the DNS name or IP address of the machine running the SAVE server. Note this needs to be entered even if the SAVE server is running on the local machine.

Notes:

1) Occasionally, the CORBA protocol has problems locating and executing servers. A fix for this is to locate the Hosts file and make an entry for the client and/or server IP address/machine name. On a Windows NT platform this is located in the Winnt/system32/drivers/etc directory. This file is used by operating systems to resolve machine names without using DNS lookups.

4.1.1.6 Miscellaneous

This section contains additional data that may be useful to a developer in supporting or modifying the SAVE server software.

SAVE Architecture - The SAVE Data Model is an object-oriented representation of data that can be communicated among manufacturing simulation tools. All objects inherit from BaseObject. Many functions in the SAVE system return a generic BaseObject reference that needs to be cast to its actual object type. The BaseObject objType variable is used to hold the type to be cast to. Most objects inherit from NamedObject. This object holds a name, a description and a datetime that the object was created or modified. As a rule, any object that can be used directly is derived from NamedObject. Any object that is not logical in its own context is referenced only within its parent object and derives from BaseObject. NamedObjects can be created and BaseObjects are created automatically by their parent objects.

Object creation New client objects cannot be referenced directly, they must be created by the DbAccess CreateObject function. This allows the server a chance to do some preliminary work like put the object reference on the memory stack and tie in the NamedObject/BaseObject inheritance references.

Sequences - Groups of SAVE objects are stored in an ObjectSeq. These sequence objects are mapped closely to ObjectStore list objects. There are two ways of manipulating sequences, either through the ObjectSeq methods or through arrays. By using arrays, a client can retrieve a stack of objects in 1 call and manipulate through them without invoking any CORBA calls.

Memory Management - When an object is created or referenced in a client, a duplicate object is created in the server. These server objects must stay in scope for the duration of client's life cycle, since it is unknown by the server when a referenced object is no longer needed. To accommodate memory cleanup each object in the server is placed on a stack. When there are no open database transactions and no client connections, the server pops all the objects off the stack and calls their destructors. The server keeps a running count of both open connections and open transactions.

4.1.2 System Administrator's Guide

4.1.2.1 Installation Instructions

The following outlines the steps needed to install and run a SAVE server.

1. Install ObjectStore 5.1

2. Install ORBIX 2.3c

3. Install the server files into a SAVE directory.

4. Start the ORBIX daemon, either from the Windows groups icons or by typing 'orbixd' from a DOS prompt.

5. Register the SAVE server with the daemon, from a DOS shell, type: 'putit save c:\temp\save\server.exe' (assuming c:\temp\save is save dir)

6. Edit the SAVE server user rights, either from the ORBIX Server Manager GUI or from the DOS prompt by typing 'chmodit save i+all' and 'chmodit save l+all'

7. Execute the SAVE server by typing 'server' from a DOS shell navigated to the save directory.

In some installations, additional steps might be required:

- CORBA is built upon TCP/IP and can use either a machine's IP address or the machine name to resolve server bind references. When referencing machines in other domains a client and/or server can get confused on a reference and not establish a connection. This can be fixed by adding a mapping of the machine name and IP address to Hosts.file. This file is found on NT in the winnt/system32/drivers/etc directory. The windows networking scheme tries to resolve machines/IP addresses there before invoking a domain name server.

- Sometimes an ObjectStore installation or setup tries to locate databases in an OS standard reference directory. The SAVE server creates and expects the database to be in the directory where the server is running, therefore this will generate an error. To fix this the OS function ossetasp needs to be invoked to change the path reference of the database as follows:

    - ossetasp executable databasepath

4.2 WFM

The SAVE Work Flow Manager (WFM) manages work flow process models consisting of processes, tasks and activities. WFM is a Java application that uses CORBA to communicate with the manufacturing simulation tools (referred to as Simulation Servers) called for by each work flow activity. The following Figure 4-1 shows the WFM and how it interacts in the CORBA universe.

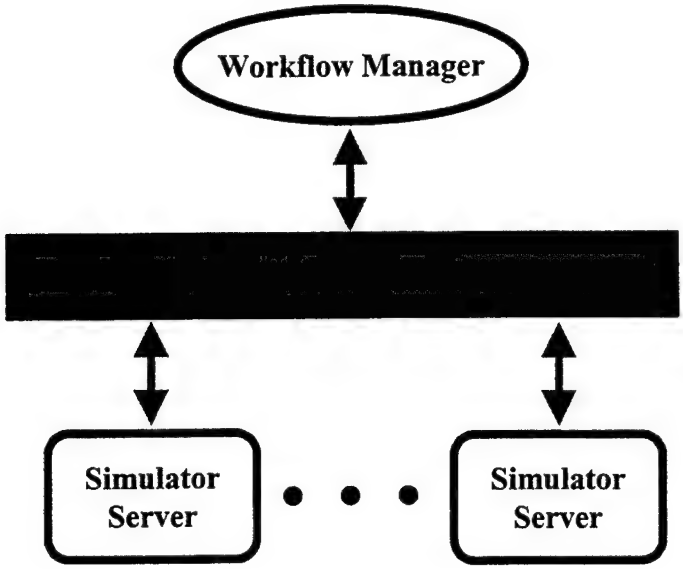

Figure 4-1 WFM Overview

Each simulator is considered as a single activity by the WFM. The WFM has a graphical interface that allows the user to create process models, execute the models and monitor their status. Currently, work flow information is stored in files, but in later versions this information will be stored in the SAVE Data Model Server.

4.2.1 Work Flow Manager Design

The SAVE Work Flow Manager (WFM) manages work flows consisting of processes, tasks and activities. It creates, saves, loads and executes the work flows that have been defined by users through a graphical programming interface.

The following Figure 4-2 shows the major components of the WFM.

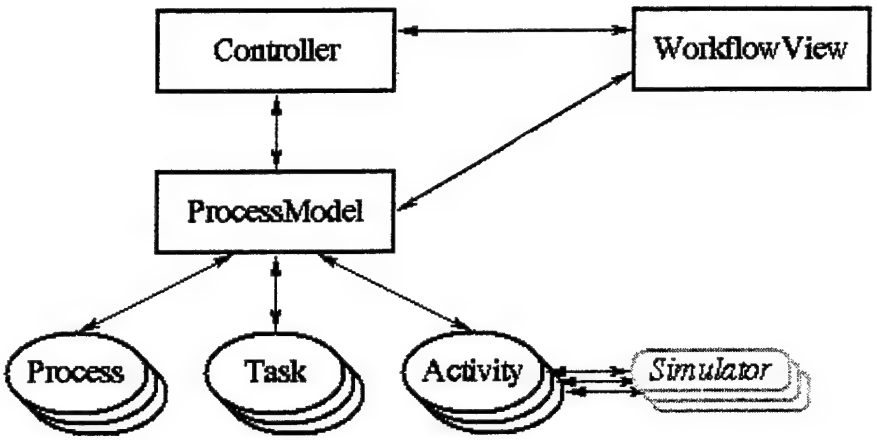

Figure 4-2 WFM Components

The following is a brief description of each component.

- **Controller** - Controls the work flow manager, including the graphical user interface. It contains a **ProcessModel**, which encapsulates the work flow, and the **WorkflowView**, which displays the work flow layout. Process Models are currently stored in files, but could be stored in the SAVE Data Model in future versions.

- **WorkflowView** - Displays the work flow objects in a layout panel. The user interacts with the objects to create work flow models.

- **ProcessModel** - Manages the **Process**, **Task** and **Activity** objects. The **WorkflowView** uses this as its display model while the **Controller** uses it to save and restore work flows to files.

- **Process** - Represents a single work flow process. It references its input, predecessor and dependent objects. A process's state is dependent on it tasks and other processes.

- **Task** - Represents a single work flow task. It references its input, predecessor and dependent objects. A task's state is dependent on its activities.

- **Activity** - Represents a single work flow activity. It references its input, predecessor and dependent objects. An activity's state is dependent other activities and its *Simulator* object.

- **Simulator** - CORBA object that serves as the proxy for a remote simulator server.

There is no single control loop in WFM; everything is done using the Java and CORBA event systems. There are a variety of Java interfaces that allow the various components to communicate with each other. In addition, there are listeners for the CORBA *Simulator*

objects so they may interact with the WFM using events. The WFM class diagram is shown in Figure 4-3.

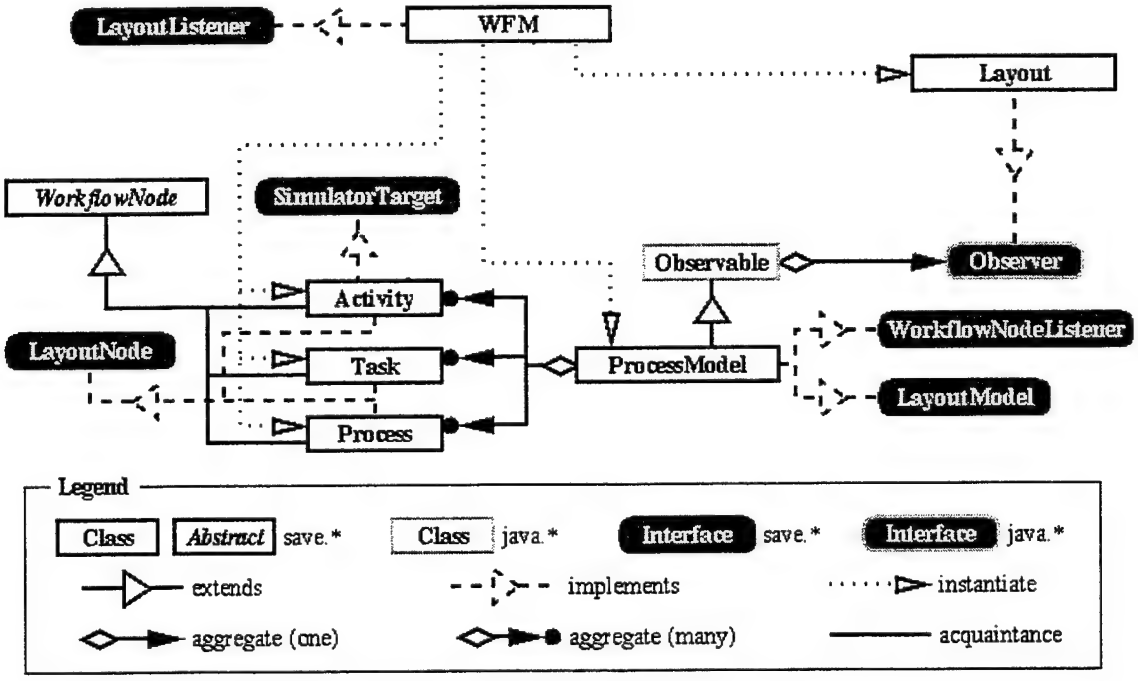

Figure 4-3 WFM Class Diagram

4.2.2 WFM Classes

The following is a brief description of each class and interface. They are grouped by packages.

4.2.2.1 save.wfm.WFM

This package is the controller and user interface for the application. Upon startup, it reads in a file that contains the *Simulator* objects. A future version could use the *Naming Service* or do a lookup in the Data Manager to get the simulators.

The controller creates a **Layout** object and registers itself as a **LayoutListener** to receive user interaction events. It then creates a **ProcessModel** object that receives state change events. Finally, it registers the **Layout** observer with the **ProcessModel** observable.

As the user constructs a model, the **WFM** creates **Process**, **Task** and **Activity** objects and places them in the **ProcessModel**. When the user executes a model, it invokes the execute method on the appropriate **WorkflowNode**. Messages and errors are returned via the **ProcessModel** nodes to the WFM for display to the user.

4.2.2.2 save.util.DataManager

This package contains the methods required to save and restore process models to/from files. This section of code was included to make it easier in future versions to handle conversion of **ProcessModel** objects to the format required by the *Data Manager*.

4.2.2.3 save.wfm.ProcessModel

This package is the observable container for the model components. It maintains a vector of **WorkflowNode** objects which are the **Process**, **Task** and **Activity** objects. In addition, the vector contains additional **LayoutNode** objects on the end, which display the node interconnections on the **Layout** display. In other words, the vector is both the work flow model and layout model. The process model implements the **LayoutModel** interface that the **Layout** uses to update itself, and the **WorkflowNodeListener** interface so it may register itself as a listener to every **WorkflowNode** object. Whenever there is an event from one of the objects, it notifies its observers (in this case **LayoutModel**) of the need for a display update.

4.2.2.4 save.wfm.Process

This package represents a single process. It implements the **WorkflowNode** abstract class and **LayoutNode** interface for interaction with the **ProcessModel** and **Layout**, respectively. It implements the draw method of **LayoutNode** so it can draw itself on the layout display.

All state changes are sent to its registered **WorkflowNodeListener** objects, which in this case are the **ProcessModel** and its predecessor node, respectively.

4.2.2.5 save.wfm.Task

This package represents a single task. It implements the **WorkflowNode** and **LayoutNode** interfaces for interaction with the **ProcessModel** and **Layout**, respectively. It implements the draw method of **LayoutNode** so it can draw itself on the layout display.

All state changes are sent to its registered **WorkflowNodeListener** objects, which in this case are the **ProcessModel** and its predecessor node, respectively.

4.2.2.6 ***save.wfm.Activity***

This package represents a single activity. It implements the **WorkflowNode** and **LayoutNode** for interaction with the **ProcessModel** and **Layout**, respectively. It implements the draw method of **LayoutNode** so it can draw itself on the layout display. In addition, it implements the **SimulatorListener** so it may receive state changes from its remote simulator.

When an **Activity** starts, it binds to its assigned CORBA *Simulator* object. It uses **CorbaUtil** to create a CORBA *SimulatorListener* object and adds itself as a **SimulatorTarget** to the object so it can receive events from the listener. It then adds the *SimulatorListener* object to its *Simulator* object so when the simulator generates an event, **Activity** receives it via the *SimulatorListener*. Finally, **Activity** starts the simulator.

All state changes are sent to its registered **WorkflowNodeListener** objects, which in this case are the **ProcessModel** and its predecessor node, respectively.

4.2.2.7 ***save.wfm.WorkflowNode***

This abstract class implements the methods required by a single node in the process model. It contains methods to get and set the state and handle **WorkflowNodeListener** objects. The class is abstract to prevent its instantiation.

The work flow node manages four data vectors. The predecessor vector contains references to objects that it controls. The successor vector has objects that control it. The input vector contains objects that must complete their execution before this node can start while the output vector has references to objects that rely on this one to start their execution. Although this sounds like duplication of information between objects, it is necessary to prevent traversing the process model to search for references when deleting objects from the work flow. Since each node carries all the information about its interconnections, it is easy to just go to those objects and perform the cleanup.

4.2.2.8 ***save.wfm.WorkflowNodeListener***

This package implements the listener methods required for events from the **WorkflowNode**. It contains a method for notifying the **ProcessModel** to update its observers.

4.2.2.9 ***save.util.CorbaUtil***

This utility class encapsulates the vendor-specific CORBA calls. All its methods are static. It is able to bind to an ORB and (eventually) the naming service, look up CORBA objects, cast CORBA objects from one class to another and create CORBA

objects.

4.2.2.10 ***save.util.SimulatorTarget***

This is the interface for any class that acts as the event recipient for a *SimulatorListener* object created using the **CorbaUtil** class. It mirrors the methods that the *SimulatorListener* object invokes in response to events from a *Simulator*.

4.2.2.11 ***save.gui.Layout.Layout***

This package is an observer object that displays the **LayoutModel** in a graphical format. The user can add and remove objects, connect the objects together, select objects and monitor a model's execution. All user interactions are sent to the WFM, which determines what to do with them. Changes in the **ProcessModel** (which implements the **LayoutModel** interface) are received via the observer interface so the display stays synchronized with the model.

4.2.2.12 ***save.gui.Layout.LayoutModel***

This package implements the observable model required by a layout observer. It requires a vector of **LayoutNode** objects and has methods for manipulating those objects.

4.2.2.13 ***save.gui.Layout.LayoutNode***

This package implements the methods required by a single node in a layout observer. It contains the methods to set and get its location and size, draw itself, determine if it is within a given point and return a list of relative objects. In this case, the relative objects are the interconnections.

4.2.2.14 ***save.gui.Layout.LayoutListener***

This package implements the listener methods required for events from a layout observer such as **Layout**. It contains the methods for mouse movements, mouse clicks and **LayoutNode** selections.

4.2.3 Interactions Between Client and Servers

The objects within the WFM communicate with the various simulators. This section

details the major interactions between the WFM internal objects and the external simulators. The interactions are presented in the following diagrams. Even though only one process and task are shown in the diagrams, there can be many processes and tasks in a work flow. Reference is made to the other objects where appropriate.

In the diagrams, text between quote marks indicate a state change while text followed by parentheses represent an action. The curved lines within an object's activation period means there is some delay (possibly manual) during that interval.

The **Starting a Process** diagram, Figure 4-4, shows how a work flow proceeds from start to end. It is always initiated from the WFM. In this case, there are no pauses, terminations or faults in the process. The other diagrams show the handling of these types of actions.

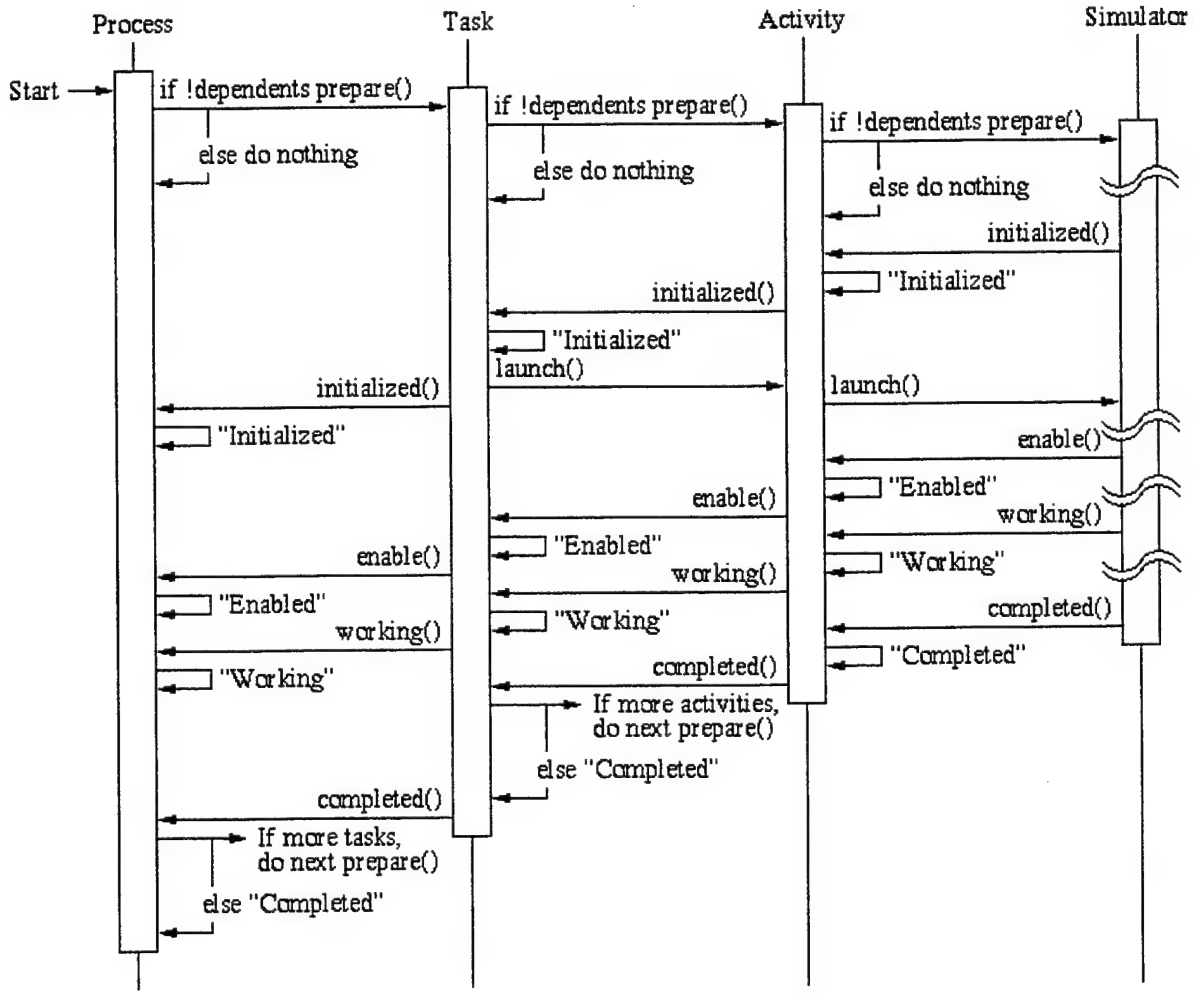

Figure 4-4 Starting a Process

The **Pausing a Working Process** diagram, Figure 4-5, illustrates the steps in pausing and resuming a process node from the WFM. If a simulator initiates the action, then start from the simulator node side.

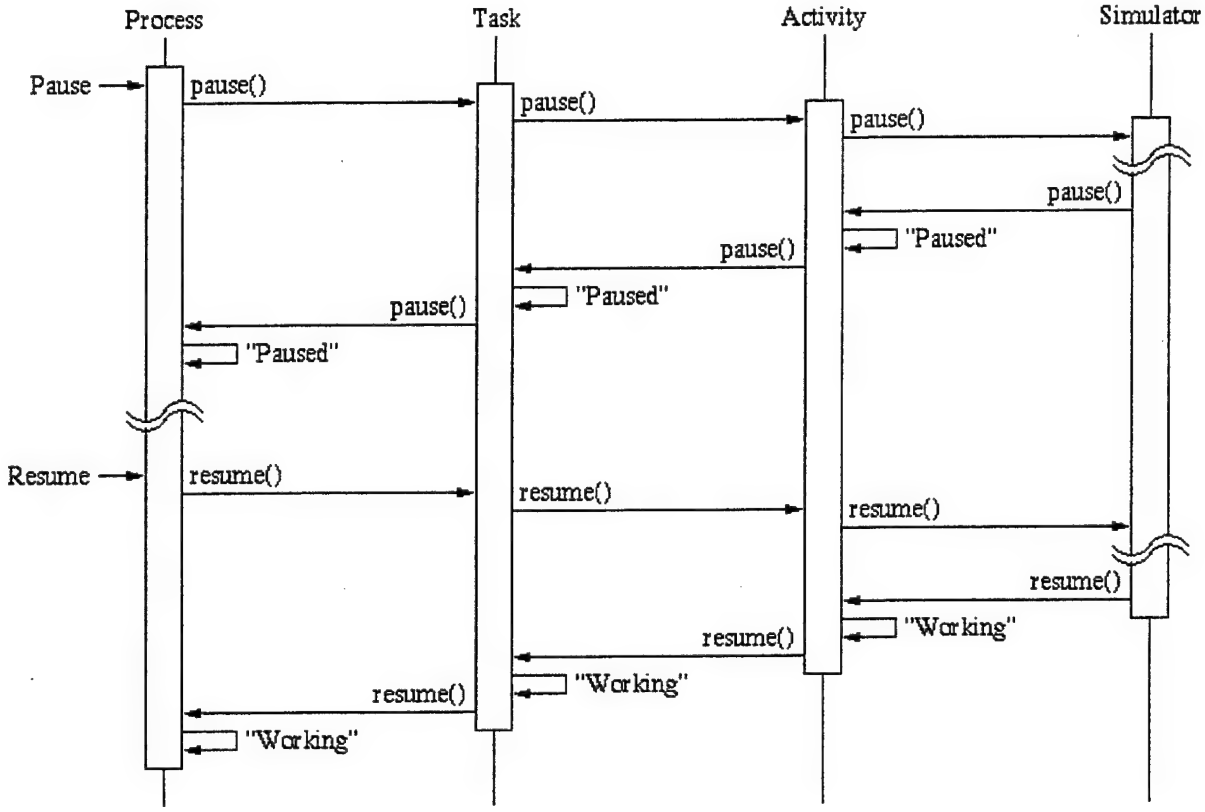

Figure 4-5 Pausing a Working Process

The **Terminating a Working Process** diagram, Figure 4-6, shows the steps in terminating a process node from the WFM. If a simulator initiates the action, then start from the simulator node side. Note that a terminate does not stop any processes that are running in parallel.

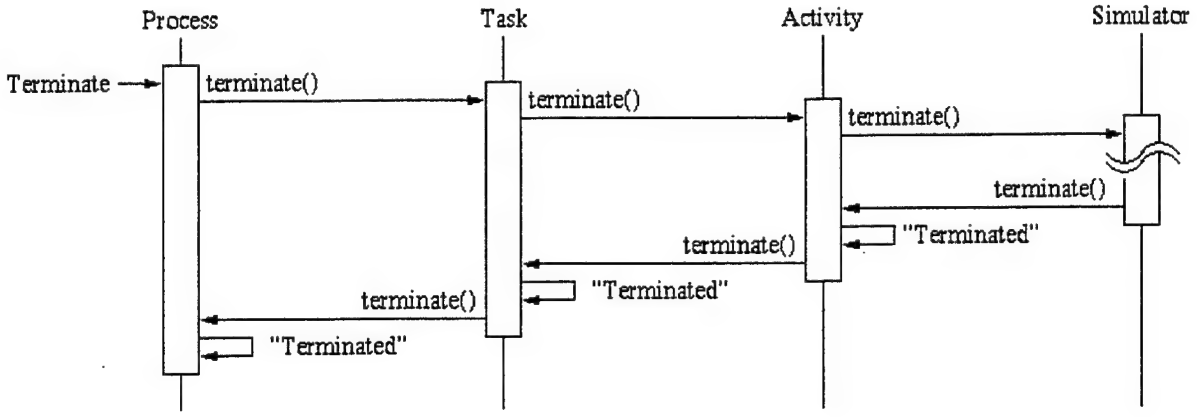

Figure 4-6 Terminating a Working Process

The **Handling a Simulator Failure** diagram, Figure 4-7, shows what happens when a simulator generates a fault. The WFM never generates a fault event; only a simulator is capable of that. Note that a fault does not terminate any processes that are running in parallel.

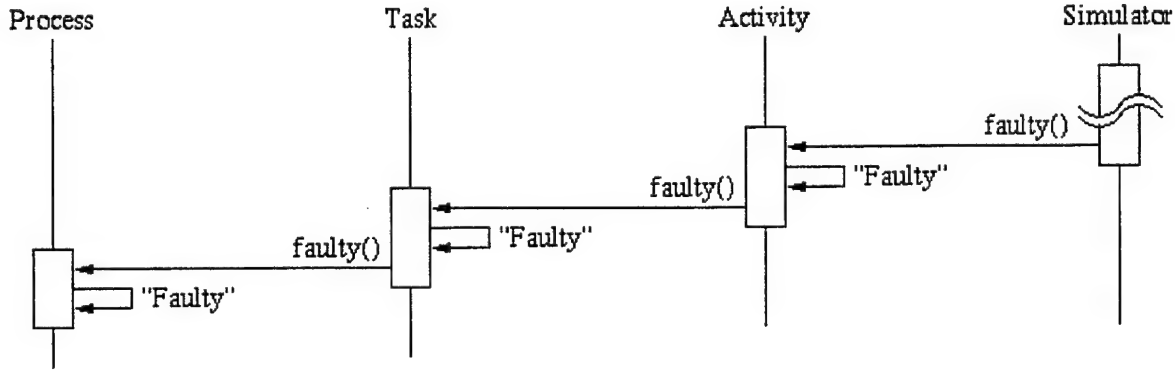

Figure 4-7 Handling a Simulator Failure

4.2.4 Work Flow Manager User Interface

The SAVE Work Flow Manager (WFM) uses Java to create the graphical user interface. The proposed interface presented here is for the July delivery. Certain menu selections will not be active in the July version but must wait until next year. These selections are present but disabled in the menu bar.

4.2.4.1 Main Window

The following Figure 4-8 shows the main window for the work flow manager:

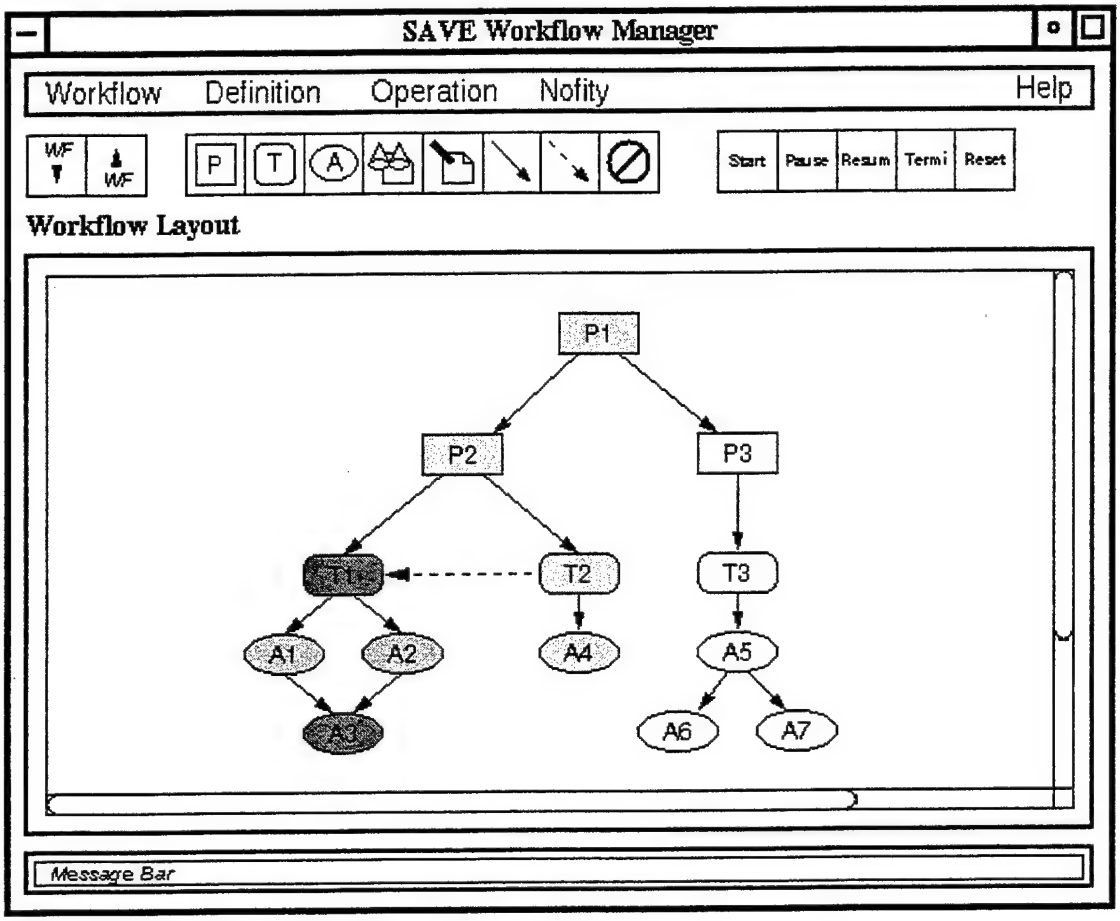

Figure 4-8 WFM Main Window

There is a menu bar at the top of the window with a tool bar that implements the major tasks just below it. The graphic layout section where process models are developed and monitored is in the center of the display. At the very bottom of the main window is the message bar which shows the hints, messages and user prompts.

The menu bar selections are show below in Figure 4-9. The selections that will not be available at the July demonstration are marked inactive.

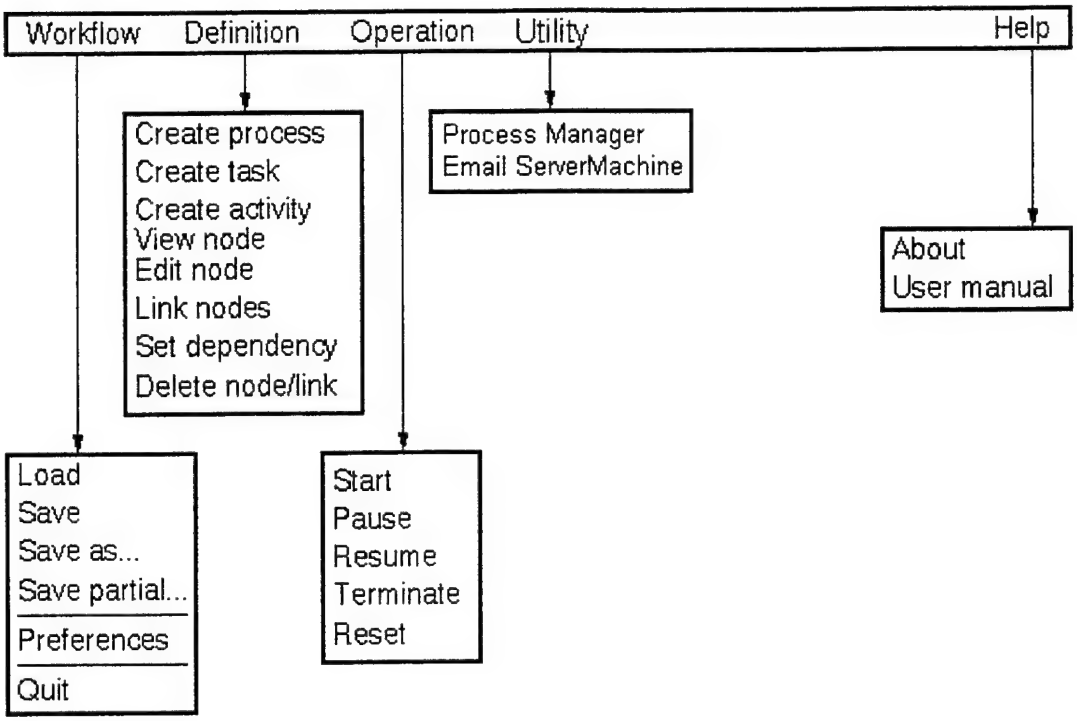

Figure 4-9 WFM Menu Bar

The menu bar breaks down as follows:

Work Flow

Load - Load a work flow model from the data manager.

Save - Save the current work flow model using the same name.

Save as... - Save the current work flow model using a new name.

Save partial... *(inactive)* -Save a selected part of the current work flow.

Preferences *(inactive)* -Set any user preferences.

Quit - Quit the application.

Definition

Create process - Create a new process node and place on the Work Flow Model.

Create task - Create a new task node and place on the Work Flow

Model.

Create activity - Create a new activity node and place on the Work Flow Model.

View node - Bring up the appropriate node viewer.

Edit node - Bring up the appropriate node editor.

Link nodes - Make a link between model nodes.

Set dependency - Make one node dependent on another.

Delete node/link - Delete a model node, link or concurrent link.

Operation

Start - Send the prepare and launch operations to the selected node.

Pause - Send the pause operation to the selected node.

Resume - Send the resume operation to the selected node.

Terminate - Send the terminate operation to the selected node.

Reset - Reset the states in the WFM to undefined so a model may be run again.

Utility

Process manager - Sender or WFM generated email message.

Email Server machine - Location of Email service machine .

Help

About - Show the about Work Flow Manager message.

4.2.4.2 Process Template

Figure 4-10 below shows the interface to create a new process. The **Subprocess/task** is a list of each subprocess and subtask and their predecessors.

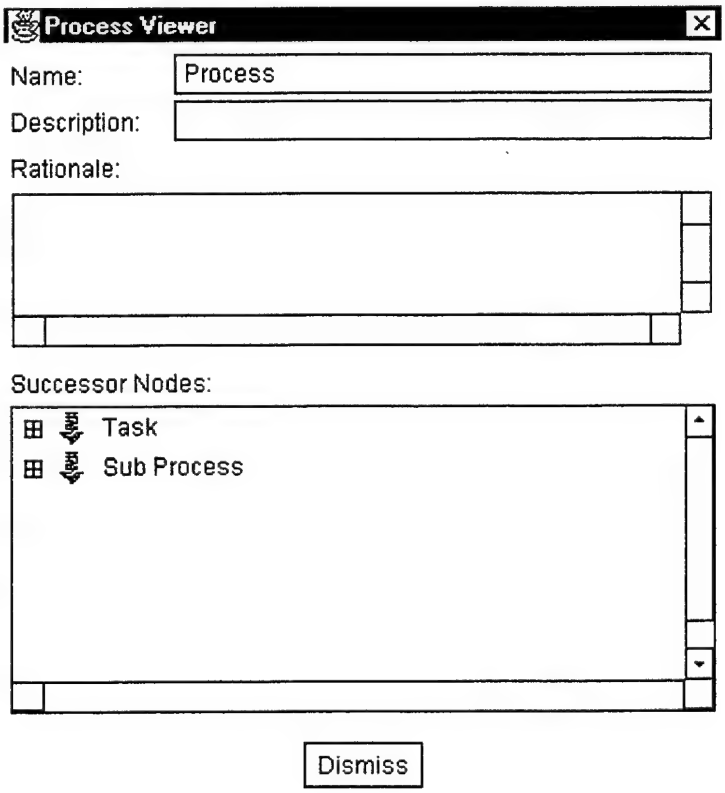

Figure 4-10 Process View Window

4.2.4.3 Task Template

The Figure 4-11 below shows the interface to create a new task. The **Activity** is a list of each activity in the activity network and their predecessors.

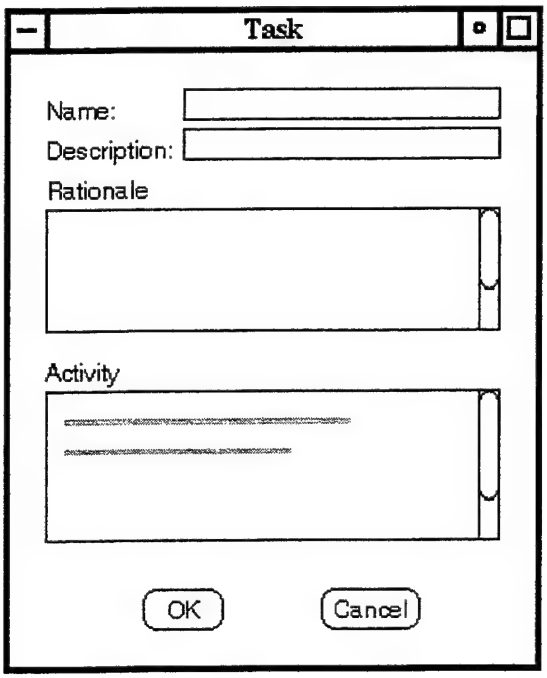

Figure 4-11 Task Window

4.2.4.4 Activity Template

The Figure 4-12 below shows the interface to create a new activity. It has entries to set the simulator information and the location of the simulation request for the simulator.

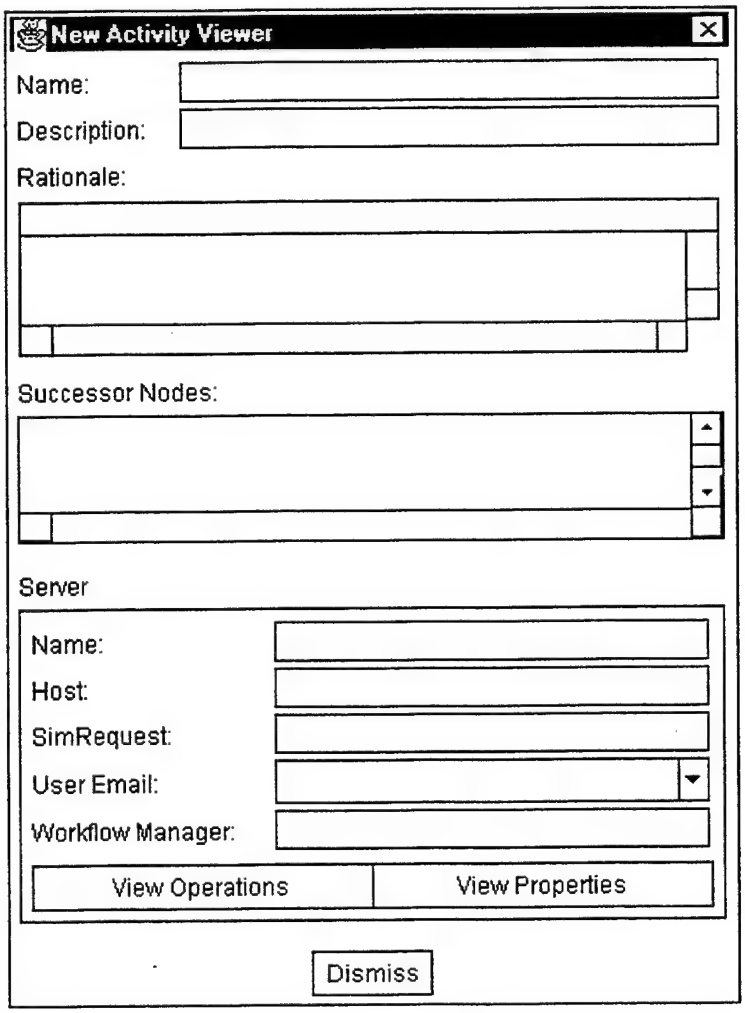

Figure 4-12 Activity Window

4.2.5 WFM Applications

The Work Flow Manager (WFM) is a Java application that manages the work flow for remote CORBA servers. The current implementation uses IONA ORBIXWeb v3.0.

Because of a bug in the ORBIXWeb 3.0 ORB, any Java servers that want to do remote testing must have a transient IOR. The tester will not bind to a persistent IOR Java server and will more than likely hang the tester. It is unknown if this bug will affect ORBIX persistent IOR servers.

4.2.5.1 Requirements

The WFM application is distributed in both **gzip/tar** format for UNIX systems and **zip** format for PC systems.

Since the test applications are in Java and require ORBIXWeb as the ORB, the following must be installed on the host machine:

Java Development Kit v1.1.6 (JDK)

Downloaded from http://java.sun.com:80/products/.

IONA ORBIXWeb v3.0

Contact IONA about getting software and a license at http://www.iona.com/.

4.2.5.2 Installation

First, follow the instructions for installing the JDK and ORBIXWeb packages. Once this is done, place the archived distribution in the directory where you wish to place the work flow manager applications. Expanding the archive creates the directory `wfm`, which contains the following:

`*.html`	-	Web page documentation.
`IDL/`	-	The IDL documentation.
`java/save.jar`	-	The Java Archive (JAR) file.
`java/server.data`	-	The WFM server configuration file.
`java/doc/`	-	The WFM documentation.
`java/doc/example/`	-	The example simulator server documentation.

4.2.5.3 Configuration

After installing all the software, configuration requires setting the paths to the ORBIXWeb `ORBIXdj` daemon and Java interpreter `java` and setting the **CLASSPATH** variable to point to the required Java classes. The files required in **CLASSPATH** are as follows:

```
<java>/lib/classes.zip (Not required for Windows 95/NT)
<ORBIXweb>/classes
<save>/java/save.jar
```

where:

`<java>` is the directory of the JDK

`<ORBIXweb>` is the ORBIXWeb directory

`<save>` is the SAVE `distrib` directory

4.2.5.4 Applications

The distribution contains the WFM and two example applications. The first example is a simulator server that implements the `Simulator` IDL. The second is a simulator tester that binds to a simulator object and manipulates it.

The first thing to do is run the ORBIXWeb ORB as follows:

```
ORBIXdj -u &
```

Once the ORB is running, you can run any of the three applications that follow.

- **Simulator Server**

  The simulator server is an example server written in Java. Run the simulator server application like so:

  ```
  java save.example.SimulatorServer [name] [type] &
  ```

  where `name` is the optional name of the simulator and `type` is the optional type. If the name is not specified, the simulator defaults to the name `Simulator` and the type defaults to 0. There are three types of simulators numbered from 0 to 2. A type 0 simulator implements all the operations. A type 1 simulator does not implement the pause and resume operations. A type 2 simulator does not implement the prepare operation. For example, to run three simulator servers named `Sim0`, `Sim1` and `Sim2`, do the following:

  ```
  java save.example.SimulatorServer Sim0 0 &
  ```

```
java save.example.SimulatorServer Sim1 1 &

java save.example.SimulatorServer Sim2 2 &
```

All the types have two commands for the terminate operation called "exit" and "kill". Using the "kill" command in a terminate operation forces an operation failed exception from the server.

When a server receives a prepare, it returns an initialized state change. When it receives a launch operation, it sends back the enabled and working state changes. To complete the work, you must change the state from the server using the state change controls. You can force a simulator state change or send a message from the simulator server using the controls on the bottom of its interface. To change the state, select the state from the choice menu and press the **From Simulator** button. Although the other controls work, this is the only one required by the SAVE program.

To quit the application, select the **Quit** button located at the bottom of the interface.

- ## Simulator Tester

The simulator tester allows testing a simulator using a low-level control panel. The tester contains controls for every method in the `Simulator` IDL. You run the simulator tester application as follows:

```
java save.tester.SimulatorTester &
```

The simulator tester user interface should appear with a message about locating the ORB. Assuming you have the Simulator Tester window displayed on your host, you must now tell it where to find your server. Enter the server host in the **Host Name or IP Address** field and the server name in the **Simulator Server Name** field and select the **Connect to Server** button. To disconnect from a server, just select the **Disconnect from Server** button. You can disconnect and reconnect as often as you like, which allows you to bring down your server, work on it, bring it back up and test it again.

If the connection succeeds, a message appears to this effect in the simulator tester's status window. At this point, you can exercise the simulator server using the controls on the left side of the simulator tester interface. All messages appear in the text window on the right side of the display. Since there is a control for every method in the `Simulator` IDL, you can manually exercise all aspects of the server.

To quit the application, select the **Quit** button located at the bottom of the interface. It is a good idea to disconnect the simulator server from the simulator

tester before closing down the simulator server. Since you can keep reconnecting to a simulator, you can bring a simulator server up and down during development while leaving the simulator tester up all the time.

- **Work Flow Manager**

The WFM takes two optional arguments. To start the WFM, enter the following:

```
java save.wfm.WFM [servers] [models] &
```

where `servers` is the optional path and file name for the list of available servers and `models` is the default directory for storing the work flow models. If `servers` is not specified, the WFM uses the default "`server.data`" file located in the current directory. If the models are not specified, they too will go in the current directory.

The servers file is a text file that contains entries for each known simulator and its IOR. A server's IOR stays the same as long as its is a named server, does not change its IDL and runs on the same host machine. The example `server.data` file included in the distribution contains four entries; one for a default simulator server and three for simulator servers that were named from the command line . The servers file is a temporary feature and will be replaced once it is possible to read the available servers from the CORBA naming service.

The format for the servers file is as follows:

```
simulator-name host-string
```

where `simulator-name` is the server's registered ORB name and may not include spaces and the `host-string` is the name or IP address of the server's host computer. You can create multiple server data files and specify them on the WFM command line.

Saving and restoring work flow models currently uses the native file system and Java serialization. This is also a temporary solution until the SAVE data manager system is capable of storing work flow model information. Since these files use serialization, they cannot be edited with a text editor.

After starting the WFM, a single window with a menu bar, tool bar, layout area and status line appears on the display. You can interactively build and run work flows on the WFM. For more information on using the WFM, refer to the Work Flow Manager User Manual in the SAVE Software User's Manual Report.

4.3 SAVE Query Manager

4.3.1 Programmer's Guide

4.3.1.1 Map to Code Organization

The Query Manager (QM) is built with the Java programming language and is divided into three sections/packages. The save/qm package references code that is peculiar to the Query Manager. It consists of the QM.Java source file and icons for the toolbar. The Query Manager references many general-purpose classes within the save.gui and save.util packages. The save.gui package contains routines for drawing GUI objects. The util package contains routines for interfacing with CORBA and data files.

There are approximately 35 SAVE objects that can be viewed and edited in the Query Manager. Each of these objects has a Frame and Panel class for displaying its contents. All of these classes are found in the save/gui package and inherit from the ApplFrame and ApplPanel classes respectively. The Frame classes instantiate in the panel class and the panel class incorporates all the drop-down boxes, textfields, etc. needed to display edit or add data to the SAVE server objects.

All of the code needed to interface with CORBA is inside the save.util package. This is in an attempt to make the code as non-ORB specific as possible. In particular the DataManager class is used to interface with the DbAccess methods in the IDL.

4.3.1.2 Required COTS Products to Modify / Run the SAVE Data Model Server

Java Development Kit, JDK1.1.6

ORBIXWeb 3.0 for WINNT/Win95

Cygnus Development Kit, GNU-Win32 tools, any version

4.3.1.3 Implementation Approach

General Application layout

The function of the Query Manager is to view, create and manipulate objects in the SAVE server's persistent data store. This is done through a set of 're-usable' SAVE objects that are stored in libraries. The Query Manager has two main windows, one contains a list of libraries and their other contains a detailed view of selected library objects.

To implement this in the Java code, four actual panels are created when the Query Manager starts up. The first is the Application Panel, which is the main panel for the applet and a container for the other three panels. The Tool Panel is at the top of the application panel and holds all of the tool icons and the code for reacting to icon events.

The Tree Panel is in the lower left and builds and displays a tree list of SAVE server libraries. When a particular library is expanded it retrieves the names of the objects in the particular library and displays them in the tree structure. The Viewer Panel is located in the lower right of the application panel. When a particular object in the library tree is selected, all of the information for that object is retrieved from the SAVE server and the viewer panel is built and populated with that data.

CORBA Connectivity

When the Query Manager starts, it reads the location of the SAVE server from a file and makes a CORBA connection. It then calls a utility routine to start a read transaction with the server. As library objects are selected, data will be read by the server and added to the Query Manager window. The viewer panel will remain blank until an actual library object is selected/highlighted.

If objects are created in the Query Manager, a persistent update transaction will be nested inside the (always open) read transaction. The update transaction closes as soon as the data is written so as not to keep a lock on the database. When the Query Manager is closed the read transaction will end.

4.3.1.4 Installation Instructions / Steps to Code Compilation

1. Install the Query Manager source code. This will typically be installed along with the Work Flow Manager code, since they share the same utility classes.

2. Install the JDK. For the current version on the Query Manager JDK1.1.6 is used, this can be downloaded free from Sun at http://java.sun.com.

3. Install ORBIXWeb 3.0. This is an Iona product that must be purchased. Upon installation, ORBIXweb will ask you for the location of the Java directory.

4. Install a Unix emulation package. This is required to run the makefiles that were written in UNIX and distributed to the PC. For the current implementation we are using Cygnus tools which can be downloaded from www.cygnus.com. This step is not needed if the makefiles are re-written to work in the installed environment.

5. Set the system variables. The top level SAVE Java makefiles reference a variable named 'SAVEROOT' which needs to contain the value of the installed SAVE directory. (e.g. c:\Save). The other makefiles include a CLASSPATH variable which point to all referenced SAVE classes, (eg C:\IONA\ORBIXWeb3.0\classes; C:\jdk1.1.6\lib\classes.zip; c:\Save\LMMS\classes).

6. The Java code is compiled from the makefiles. Open a DOS shell window and navigate to the Query Manager directory. Type: "make all" to compile all of the query manager Java files and type "make run" to execute the query manager. To simplify execution, the class files have been zipped up into a save.jar file and a batch file has been written to execute the query manager from within the jar file.

4.3.1.5 Miscellaneous

1) The Query Manager requires the SAVE server to be up and running since it establishes a connection immediately.

2) All classes can be zipped up into a save.jar file and the query manager can be run from it. A sample of this is found in the batch file folder. (As well as running the Work Flow Manager and Tester programs)

3) The Query Manager can access a server on the same machine or on a different one. If the Query Manager is to run on the same PC that the server is running, it is advisable to register the SAVE server with the ORBIXWeb Java daemon. This is possible since the ORBIX server daemon is a subcomponent of the ORBIXWeb Java daemon. Follow the same steps listed in the server section to register the SAVE server.

4) The Query Manager was initially created as an application and later converted to an applet. There are some problems associated with this conversion. First, the Query Manager originally read the server location from a flat file that was read in by the Java virtual machine. This is not feasible since applets are not permitted to read computer files, so the Query Manager will incorporate naming service and utilize it to locate the SAVE server machine. Another problem associated with an applet is that many of the current screens are associated to building panels inside of frames. An applet uses the browser for its overall frame and typically builds panels inside of it.

4.4 MS Project Wrapper

Microsoft Project 98 has the ability to import and export SAVE Process Plans as schedules via a CORBA compliant Visual Basic application which bridges Project 98 to SAVE. The Visual Basic application makes use of Iona's CORBA/ActiveX Bridge to supply the Project 98 to SAVE link.

The Project 98 Wrapper provides a graphical user interface into the SAVE library, from which the user may create or select Simulation Request, Design Studies, Design Study Alternatives, and Process Plans. The end result, is that a Project 98 schedule is exported into SAVE as a Process Plan, or a SAVE Process Plan is imported as a schedule into Project 98. The User interface is window driven, which guides the user through the selection process of exporting or importing.

4.4.1 Starting the Project 98 Wrapper

The Project 98 Wrapper takes only one argument. To start the Wrapper, click on the SAVE_Database toolbar button displayed at the top of the Microsoft Project 98 Gantt Chart window. Upon installation, this toolbar should have been configured with the server argument that specified the IP address and name for the SAVE server.

4.4.2 Wrapper Interface

The Project 98 Wrapper interface consists of a series of windows which prompts the user for input, advises the user of possible selections and displays status as progress is made. The primary windows are The CORBA Connection window, Initial Selection window, Simulation Request window, Import Design Study Process Plan window, Export Process Plan to SAVE window, and Process Plan Progress window.

4.4.2.1 CORBA Connection Window

The CORBA Connection window (Figure 4-13) prompts the user to enter the IP address or machine name of the machine hosting the SAVE server. A default IP address is provided at installation, but the user may override this address if desired. When the user selects next, an attempt to connect to the SAVE server is made. If the connection is successful, the Initial Selection window is displayed.

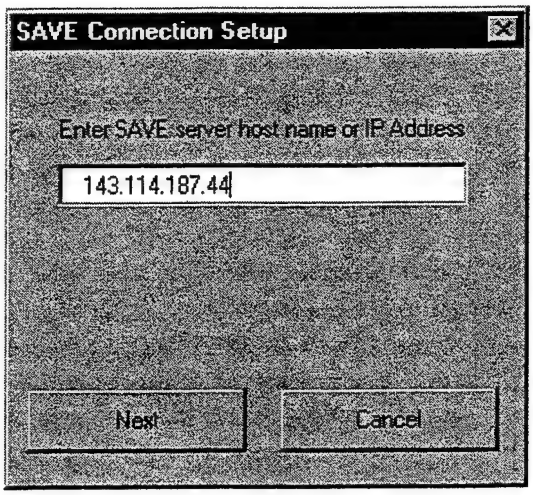

Figure 4-13 CORBA Connection Window

4.4.2.2 Initial Selection Window

Figure 4-14 shows the Initial Selection window. The user will have several options to choose from. The Import Parts List selection is intended to allow the user to import a BOM (Bill of Materials) in the form of a text file, which will be displayed as a set of tasks in MS Project. At this time, the Import Parts List option is not functional. The second option starts the process, which allows the user to select and open a Simulation Request. The third option starts the process, which allows the user to select and open a Design Study Process Plan. Finally, the fourth option starts the process, which allows the user to export an MS Project schedule to SAVE. If the Gantt chart from which the wrapper is started is blank, the export option will not be enabled. Once an option is selected, the user presses <Next> to continue. The user may also select <Cancel> to exit the Wrapper.

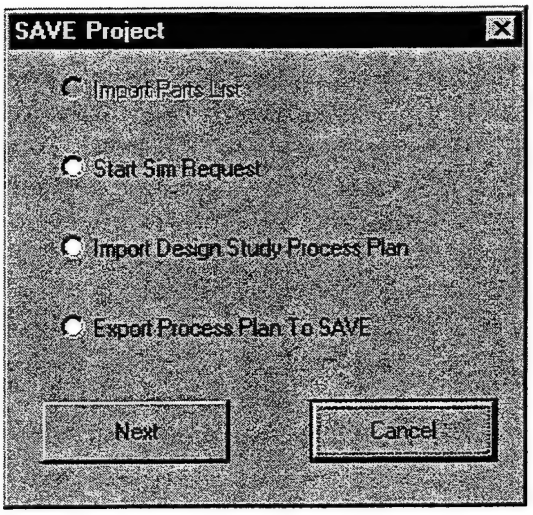

Figure 4-14 Initial Selection Window

4.4.2.3 Simulation Request Window

The Simulation Request Window (Figure 4-15) will display a list of Simulation Requests found in the SAVE library. The user may also select <New>, which will prompt the user, for the name and description of a new Simulation Request. If the SAVE library does not contain any Simulation Requests, the user will be immediately prompted to enter the name and description of a new Simulation Request. From the Simulation Request window, The user may also select <Back>, which returns to the Initial Selection Window, or <Cancel> to exit the Wrapper. If the user makes a selection and presses <Next>, the Process Plan Progress Window will appear. The Process Plan to be displayed (as a Project schedule) will always be from the Simulation Request ProcessPlan object. The only way to view an alternative process plan, is to open a Design Study via the Import Design Study Process Plan option on the Initial Selection window.

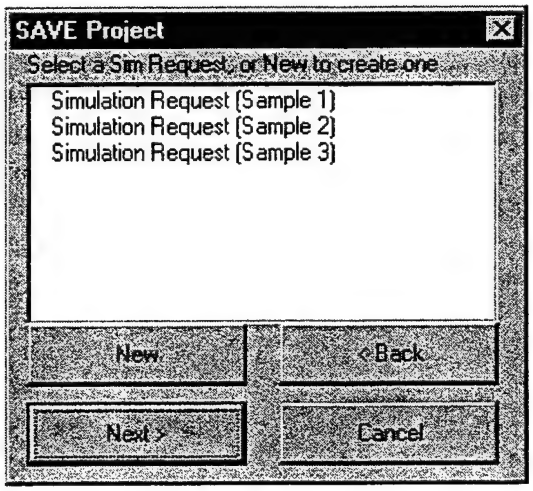

Figure 4-15 Simulation Request Window

4.4.2.4 Design Study Window

The Design Study window (Figure 4-16) appears when the user selects the Import Design Study Process Plan option from the Initial Selection window. A list of Design Studies is shown, from which the user may select. The user may also press <New>, which brings up a prompt for the name and description of a new Design Study. If the SAVE library does not contain any Design Studies, the user is immediately prompted for the name and description to create a new Design Study. At any time, the user may select <Back> to return to the Initial Selection window, or <Cancel> to exit the Wrapper. Once the user has made a selection and pressed <Next>, a Design Study Alternative must be selected. If the Design Study has both a Selected Alternative and a list of Alternatives, the user will be prompted to choose either the Selected Alternative, or to view the list of other Alternatives. If only the Selected Alternative object exist, then the list of process plans contained within it are listed, so that the user can select one. Similarly, if only the Alternative lists object exist, the list of alternatives are displayed for the user to select.

When the user selects an alternative from the list, the process plans within that object are listed. Once the user selects a process plan from the list (either from the Selected Alternative or an alternative from the Alternative list), the Process Plan Progress window is displayed to show progress as the process plan is imported into MS Project. Although the user can select and display a process plan from the Selected Alternative object, the Wrapper will not allow the user to export a Project Schedule to a Selected Alternative. All Project schedules are saved to process plans owned by Alternatives from the Alternative list. The Selected Alternative will point to one of the alternatives from the Alternative list once the team has made a final design selection from the list of Alternatives. Assigning the final selection to the Selected Alternative is accomplished with the Query Manager tool.

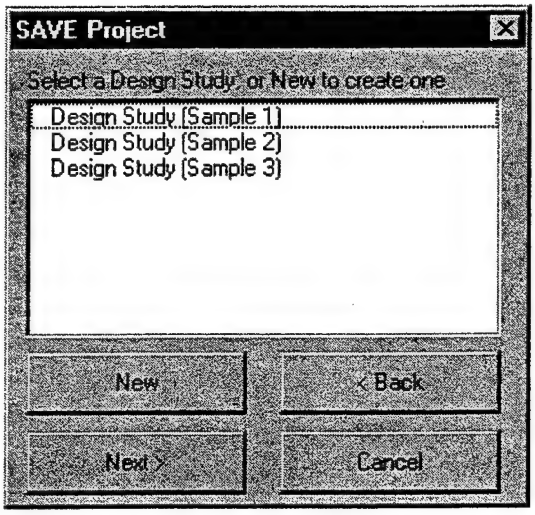

Figure 4-16 Design Study Window

4.4.2.5 Process Plan Progress Window

The Process Plan Progress window (Figure 4-17) displays status messages as a process plan is either imported or exported. The finish button is disabled until the import or export is complete.

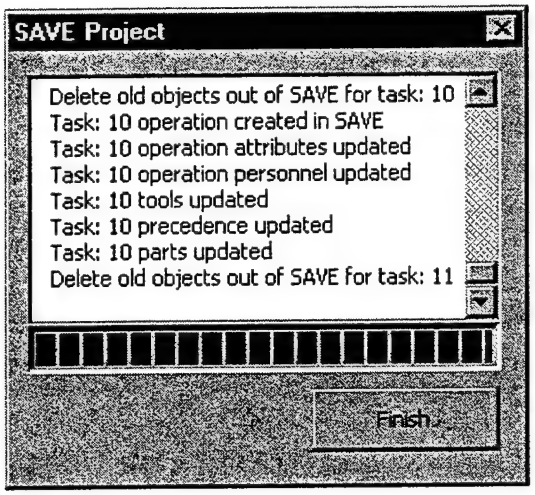

Figure 4-17 Process Plan Progress Window

4.4.3 Wrapper Installation

Follow these steps to install the MS Project wrapper :

- Under the Install Kit folder, open the CORBA COM SAVE Bridge folder and copy the files to a folder on your machine.
- If the ORBIX runtime is not loaded on your machine, you will need to install c:\IONA\ORBIX_2.3c\BIN\ioleM23C.dll
- Locate REGSVR32.exe on your machine and Note its path
- On the Start menu click Run
- In the dialog box type the following:
- <path to RegSvr32>\RegSvr32.exe C:\IONA\ORBIX_2.3c\Bin\iolem23c.dll
- On the Start menu click Run
- In the dialog box type the following:
- <path to RegSvr32>\RegSvr32.exe <path to CORBA COM SAVE Bridge files>\SaveBroker.dll
- Before installing Project 98, run a search on your machine for COMCTL32.OCX. It is generally found at C:\WINNT\system32 If found proceed to Step 5. If not found, copy the version in the ActiveX Control folder to C:\WINNT\system32. You must now register this control.
- On the Start menu, click Run.
- In the Run dialog box, type the following:
- <Path to RegSvr32>\REGSVR32.EXE <Path to OCX>\OCXFILE.OCX

  For example:

  C:\Devstudio\VB\REGSVR32.EXE
  C:\Winnt\System32\COMCTL32.OCX

  NOTE: If Regsvr32.exe is in the System or System32 folder, the path is optional.

- Next Step
  - Install Project 98. Once installed, start it.
  - From the menu bar, select Tools -> MACRO -> Visual Basic Editor.

- From the Visual Basic menu bar, select Tools -> References...
- In the Reference window, select Standard ORBIX Types. If you can't find it, select Browse then go to: c:\IONA\ORBIX_2.3c\BIN\ioleM23C.dll and open the dll. It should now be selected in the Reference window.
- Select the Browse button again, and go to the folder where you copied the CORBA COM SAVE Bridge files. Open SAVEBroker.DLL. SAVEBroker Type Library should now be checked in the Reference window. Click OK to close the Reference Dialog Box.
- Close the Visual Basic Editor

- Next Step
  - From the Project 98 menu bar, select File -> Open. Go to the Install Kit\The Wrapper folder, and open Wrapper.mpp
  - From the Project 98 menu bar select Tools -> Organizer...
  - In the organizer box, select the Modules tab. Select all the modules in the Wrapper window, and move them to the Global.mpt window.
  - Select the Toolbars tab in the Organizer box. Copy the Save toolbar to the Global.mpt window.
  - Select the Tables tab in the Organizer box. Move the SaveDb table to the Global.mpt window. Close the Organizer box.

- You are now ready to run the Wrapper. Make certain that a SAVE Server is running, and that you know the IP address of the server. Select the SAVE Database toolbar button, and follow the prompts.

4.4.4 Wrapper Software Design

The MS Project Wrapper is a Visual Basic application, which makes use of Iona's ActiveX to CORBA Bridge. The SAVE IDL is compiled using Iona's winIDL compiler, which creates visual basic objects for each SAVE CORBA object. The Visual Basic application is embedded into MS Project 98 as a macro. The application is divided into two major components. The form components are the graphical user interfaces, while the module components provide the underlying SAVE to MS Project conversions.

4.4.4.1 User Interface Forms

The wrapper makes use of the following graphical user interface forms.

- frmSAVEConnect - This form prompts the user for the IP address used to connect to the SAVE server.

- frmSAVEErrorBox - This form displays an error message, that too many Comm Failures are occurring. The User is prompted to cancel the wrapper execution, or to continue trying.

- FrmSAVEexport - This form prompts the user on export choices, based on the control state the wrapper is in.

- frmSAVEimportDs - This form, prompts the user for any Design Study inputs needed.

- FrmSAVEinit - This form is the initial page that prompts the user to choose between importing a part list, or opening a simulation request, or opening a design study, or exporting a MS Project schedule to SAVE.

- FrmSAVEpartsList - This form prompts the user to import a parts list. For now, this code is not used, waiting for requirements.

- frmSAVEprogress - This form is a progress window, which displays progress when a SAVE process plan is imported into MS Project or a schedule is exported to SAVE.

- FrmSAVESimReq - This form prompts the user for Simulation Request inputs.

4.4.4.2 Wrapper Modules

The Wrapper makes use of the following modules to translate SAVE objects to MS Project data, and vise versa.

- SAVE_Database - The SAVE_Database module provides methods to connect and disconnect from the SAVE server. Methods are also provided to retrieve library list objects from the SAVE library, as well as the list item count and list item names.

- SAVE_DesignStudy - The SAVE_DesignStudy module provides methods for accessing Design Study objects as well as objects within a Design Study object. The following is a breakdown of the subroutines and functions provided by this module.

  - ✓ CreateDesignStudy - This subroutine creates a new Design Study in SAVE, and assigns the module pointer DesignStudy to it. Next the Design Study's Alternative object is retrieved from SAVE, the module pointer DsAlternative is set to this object.

  - ✓ WrDsSaBpp - This subroutine creates and fills in the Design Study Selected Alternative Baseline Process Plan. First, a process plan is created using the name and description found in task(1) of the Project file. Once the Process Plan is created in SAVE, a pointer (SAVE_ProcPlan.ProcPlan) is assigned to the new

Process Plan. Calling SAVE_ProcPlan.WriteProcPlan fills in the SAVE version of the Process Plan.

- ✓ GetDesignStudy - This subroutine retrieves a Design Study from the SAVE library based on the index passed as a parameter. The module pointer DesignStudy is set to the object retrieved. The Design Study's Alternative list object is then retrieved, and the module pointer DsAlternative is set to this object. A flag is then set to indicate whether the list is empty or not. Next, the Design Study's SelectedAlternative object is retrieved. This object is tested, and a flag is set to indicate if the object is null. If it is not, the SelectedAlt module pointer is set to it. The SelectedAlternative's Process Plan list object is then retrieved. A flag is set to indicate if the list is empty.

- ✓ GetDsByName - This function retrieves a Design Study from the SAVE library using the name specified in the public variable DesignStudyName. To do this, the function gets each Design Study one by one from the SAVE library, until a name match is found. Once the object is found, the module pointer DesignStudy is set to the object.

- ✓ GetDsAlt - This subroutine retrieves a Design Study Alternative from the Design Study Alternative list based on the index passed.

- ✓ SetDsAlt - This subroutine sets the module pointer DesignStudyAlt to the Design Alternative object passed as a parameter. The Design Alternative Process Plan list is retrieved, and the module pointer ProcPlanSeq is set to it.

- ✓ GetDsAltByName - This function searches for the Design Study Alternative by name, using the name specified in the public variable SelectedDesignStudy. Typically, the caller sets this variable. Once the Alternative is found, the object is retrieved and the module pointer DesignStudyAlt is set to it. The Alternative's Process Plan list is retrieved, and the module pointer ProcPlanSeq is set to it.

- ✓ ReadBaselineProcPlan - This subroutine retrieves the BaselineProcPlan object and calls SAVE_ProcPlan to have it displayed.

- ✓ ReadPpsPp - This subroutine gets a Process Plan object from the list of Process Plans, and calls SAVE_ProcPlan to display the Process Plan as an MS Project schedule.

- ✓ AddDsAlt - This subroutine creates a new Design Alternative using the name found in the public module variable SelectedDesignStudy, and the description found in the public module variable DesignStudyDesc. Once the object is created, the pointer DesignStudyAlt is set to it. Finally the new Design Study Alternative is added to the list of Alternatives.

- ✓ OwrBaselinePp - This subroutine gets the BaselineProcPlan object; sets the pointer SAVE_ProcPlan.ProcPlan to it, and then calls SAVE_ProcPlan to overwrite this object with the data found in the MS Project schedule.

- ✓ WrDsAltBpp - This subroutine creates a new Process Plan, sets the SAVE_ProcPlan.ProcPlan pointer to it, and calls SAVE_ProcPlan to fill it in with the MS Project schedule data. Once finished, the DesignStudyAlt.BaselineProcPlan pointer is set to the filled in process plan.

- ✓ AddDsAltPpsPp - This subroutine adds the process plan pointed to by the SAVE_ProcPlan.ProcPlan pointer to the list of Process Plans pointed to by the module pointer ProcPlanSeq.

- ✓ OwrDsPpsPp - This subroutine gets the indexed Process Plan object from the list of Process Plans. Sets the SAVE_ProcPlan.ProcPlan pointer to it, and then calls SAVE_ProcPlan to overwrite it with new data from the MS Project schedule.

- ✓ CreateAddPpsPP - This subroutine creates a new Process Plan, sets the pointer SAVE_ProcPlan.ProcPlan to it, fills it in with data from the MS Project schedule, and finally adds the filled in process plan to the list of process plans pointed to by the module pointer ProcPlanSeq.

- ✓ GetAltSeqCount - This function returns the number of Design Study alternatives listed in the Design Study Alternative list.

- ✓ GetDsAltName - This function returns the name of an indexed Design Study object from the list of Design Study objects.

- ✓ GetSelBlPpName - This Function returns the name of the selected Baseline Process Plan. The SelectedAlt module pointer is used to retrieve the BaselineProcPlan object. The name of the BaselineProcPlan object is returned.

- ✓ GetAltBlPpName - This function returns the name of the BaselineProcPlan object pointed to by the DesignStudyAlt module pointer. The DesignStudyAlt pointer points to a Design Alternative from the Design Alternative list.

- ✓ GetPpCount - This function returns the number of Process Plans listed in the list of Process Plans.

- ✓ GetPpName - This function returns the name of a indexed Process Plan from the list of process plans.

- ✓ SetDesignStudy - This subroutine takes the Design Study object passed to it, and sets the module pointer DesignStudy to it. The DesignStudy Alternative list object is then retrieved and the module pointer DsAlternatives is set to it. A flag is set to indicate if the list is empty. Next the DesignStudy SelectedAlternative object is retrieved. A flag is set to indicate if the object is empty. If the Object is not empty, the SelectedAlternative Process Plan list object is retrieved. A flag is set to indicate if this list is empty.

- SAVE_ErrorHandler - The SAVE_ErrorHandler provides error-handling capability. Visual Basic does not have Try/Catch statements, which you would normally use

when executing CORBA commands. Instead, each method which has a CORBA command, has a ON ERROR GOTO ERRORHANDLER statement. Each ERRORHANDLER in turn calls this Function. This function will examine the Err.Number to determine if the error ISCORBA related. If the error is CORBA, then function GetCorbaError is called to return a string description of the error. Individual case statements handle the exceptions that the SAVE server throws. All other CORBA errors are handle by a Case-Elsе statement.

- All errors are fatal, with one exception. CORBA COMM_FAILURE is not fatal. This error generally occurs when a CORBA command times out before completing. In this case this function returns a true flag, to indicate the CORBA command should be tried again.

- COMM_FAILURE requires special handling. We do not want to get into an infinite loop re-trying. A timer is used to time how far apart the COMM_FAILURE are occurring. Also a counter keeps track of the total number of COMM_FAILURE. In the event that a COMM_FAILURE occurs after both read and write transactions to the server are closed, then the failure occurred during the closing of the read transaction. In this case we should just exit, and not attempt to re-close the read transaction.

- SAVE_Globals - This module holds all the global variables, which are used by the other SAVE modules.

- SAVE_Main - This module is the main starting point for the SAVE wrapper, and the final ending point once the wrapper is complete.

- SAVE_ProcPlan - The SAVE_ProcPlan module provides methods for accessing Process Plan objects as well as objects within a Process Plan object. The following is a breakdown of the subroutines and functions provided by this module.

✓ DisplayBlankPP - If a user starts the SAVE wrapper from a blank project file, and creates a Sim Request or Design Study, this subroutine will add the new Process Plan name and description to the first task line of Project.

✓ DisplayProcPlan - This subroutine is called to display a Process Plan retrieved from the SAVE server. The calling program must set the public module variable ProcPlan to the Process Plan the SAVE server retrieves prior to calling this subroutine.

✓ AddTask - The AddTask subroutine creates a task in Project for each Operation within a Process Plan. All relevant Project tasks data is mapped to its corresponding SAVE Operation attributes.

✓ ConvertSaveDate - This function is used to convert the SAVE date format to a format recognizable by MS Project.

- ✓ ConvertProjectDate - The function is used to convert the Project date into a format that is recognizable by SAVE.

- ✓ WriteProcPlan - This subroutine is used to write the MS Project schedule to SAVE. Care must be taken to not destroy data that already exist in SAVE, but that is not used by MS Project. For this reason, if an Operation already exists in SAVE, we do not overwrite it, we only overwrite the fields in it that are used by MS Project.

- ✓ WriteTask - WriteTask is the subroutine that writes each tasks to a corresponding operation in SAVE.

- ✓ GetCount - This function is a utility that determines how many sub strings are in a string that is delimited by the character passed as the first parameter (delim).

- ✓ FillArray - This function is a utility that takes the sub strings in a string, and places them in an array. The size of the array is determined by the parameter "num". The delimiter character is passed in "delim".

- ✓ FindPreds - This function is a utility that takes the Predecessor string from a Project task, and extracts each predecessor, placing it in an array. Since it is difficult to determine how many predecessors are in the string, before hand, we assume no more than 20 will be allowed.

- ✓ AddPred - This subroutine adds predecessor numbers to a given task's predecessor field. If the field is empty, simply add the new number to the field. If the field already has numbers in it, then before adding the next number, add a "," as a delimiter.

- ✓ BuildToolArray - This function fills an array with the tools listed in the string passed. The function returns the number of tools listed in the array. The tool names are placed in ToolArray2. ToolArray2 is a 2-diminsional array. The first dimension holds the name of each tool, while the second dimension holds the quantity of the tool used.

- ✓ SAVE_SimReq - The SAVE_SimReq module provides methods for accessing Simulation Request objects as well as objects within a Simulation Request object. The following is a breakdown of the subroutines and functions provided by this module.

- ✓ GetSimReq - This function retrieves a Simulation Request from the SAVE database, and stores the Simulation Request object in the module pointer SimReq.

- ✓ WriteProcPlan - This subroutine creates a process plan in the SAVE database, and calls SAVE_ProcPlan to fill the process plan. Once the plan is filled, the SAVE Simulation Request Process Plan is set equal to the new process plan.

- ✓ OwrProcPlan - This subroutine overwrites an existing Process Plan. The index parameter indicates the type overwrite. If Index is -1 then overwrite to the Simulation Request's active process plan. If Index is 0 then overwrite Simulation Request

Alternative Baseline Process Plan. If Index is greater than 0 then get the process plan pointed to by the index from the Design Study Alternate Process Plan list, and overwrite it.

- ✓ CreateSimReq - This subroutine creates a new Simulation request from the name found in the public module variable SimReqName, and description found in the public module variable SimRegDesc. The new object is stored in the public module variable SimReq.

- ✓ ReadProcPlan - This subroutine reads the Simulation Request Process Plan from SAVE, and calls SAVE_ProcPlan to display it in the active Project Worksheet.

- ✓ SetNewSimDs - This subroutine sets the pointer in the Simulation Request, to point to a new Design Study. The Design Study name is held by the SAVE_DesignStudy module, and was put there by the user interface form: frmSAVESimReq. The frmSAVESimReq prompted the user for the name.

- ✓ SetNewDsAlt - This function sets the Design Study Alternative in the Simulation Request. The module variables SimReqName and SimReqDesc are set via user input by the calling frmSaveSimReq.

- ✓ SetNewSimPp - This subroutine creates a new process plan using the name and description found in the module variables SimReqProcName and SimReqProcDesc. These were set by the form frmSAVESimReq based on user input. Once the Process Plan has been created, calling WriteProcPlan fills it in, but only if the Project file actually has Process Plan data. Otherwise the Process Plan is left empty, and a one line Project File is created indicating the name of the Process Plan now in SAVE.

- ✓ GetDesignStudy - This subroutine calls SAVE_DesignStudy to retrieve the Design Study object's Alternate list, and Selected Alternate. The Design Study object is passed as a parameter. Flags are set to indicate if the Alternate List or Selected Alternate exists.

- ✓ SetDsAlt - This subroutine calls SAVE_DesignStudy to retrieve a Design Study Alternative from the Design Study Alternative list based on the index passed. The list of Process Plans from the Alternative is retrieved. A flag is set to indicate if the process plan list is empty. The Simulation Request's Design Alternative is set equal to the Alternative just retrieved.

- ✓ GetDsAlt - This subroutine passes a SimReq.DesignAlternative object from SAVE to the module SAVE_DesignStudy who in turn retrieves the Process Plans list from this Alternative object. A flag is set to indicate if the list is empty.

- ✓ GetProcPlan - This subroutine retrieves the Simulation Request Process Plan, and sets the module pointer SimProcessPlan to the process plan object.

- ✓ GetActivePpName - This function returns the name of the Simulation Request Process Plan.

- ✓ GetBlPpName - This function returns the name of the Simulation Request Design Alternative Baseline Process Plan.

5.0 Recommended Enhancements

The SAVE development team has identified a list of potential enhancements that should be considered for future versions of SAVE. The SAVE contract was tasked with developing and demonstrating the functionality of an integrated set of commercial manufacturing simulation tools, and has been successful in doing that. It was always the intent that additional effort would be required to produce robust, full-function, infrastructure elements. The ideas below present those things that the software development team identified as beneficial or needed but were beyond the resources of the existing contract.

- Additional Simulation Tool Categories
- Data Model Changes
- Configuration Management
- Distributed back-end data storage
- Use of CORBA Naming Services
- Performance Improvements

5.1 Additional Simulation Tool Categories

During the demonstration and beta testing of the SAVE system it became quite apparent that better automation of the initial creation of a manufacturing process plan would be very beneficial to users of the SAVE system. The SAVE Data Model was designed to allow new categories of tools within the manufacturing simulation problem domain to be easily added, so this should present no significant problems. The initial tools that were made SAVE-compliant were chosen to provide a wide enough range of capability to validate the flexibility of the Data Model and the development team is confident that this is the case.

The Microsoft Project planning tool was wrapped to be SAVE-compliant as part of one of the SAVE beta test site activities. This tool proved useful in inputting an existing process plan for a redesign activity as the starting point for other simulation tools. However, there are several good manufacturing process planning tools now on the market, and the SAVE team recommends that any implementation site consider adopting one of these tools to use in the SAVE environment. Initial discussions were held with several of these vendors and all expressed interest in providing SAVE-compliant versions of their tools if desired by customers.

As seen in the current SAVE architecture, the CAD tool is loosely coupled to SAVE and is not directly wrapped to share data through the SAVE Data Model. Simulation tools access CAD data either through standard geometry files (faceted geometry for visualization in IGES or STL formats) or custom links to extract cost and tolerance feature data. These access methods pre-existed SAVE and provided the advantage that any of the four major CAD tools can be used with SAVE with virtually no changes to the system (feature extraction links already exist for CATIA, IDEAS, ProEngineer, and Unigraphics). An area of SAVE expansion that has strong interest

among potential users is to directly wrap the CAD tool to access features (form features for costing , and functional features for tolerance analysis) in an industry standard way. The CAD wrapper would access this feature data from a CAD model and would populate that information into the SAVE Data Model in the Part-Feature data objects. Once in SAVE these vital data would be available to any SAVE-compliant tool without having to develop custom CAD interfaces (estimated at $200-250K development cost). The potential for SAVE-compliant CAD wrappers should be strongly considered in any future development activity.

5.2 SAVE Data Model Changes

Although the SAVE Data Model has proven to be quite robust during the demonstrations and beta test activities in Phase II of the program, there are a few areas of the model that may need to be modified to make the model more complete and easier to use.

5.2.1 Schedule

Currently, schedule objects are associated with manufacturing orders, process plans and operations. There is some concern that individual attributes within the process plan and operation objects may be duplicating this information, or at the least, creating confusion over the meaning of the values. Table 5-1 provides a list of the relevant attributes.

Table 5-1: Duration Attributes in the SDM

Object	Attribute
Schedule	Planned Duration
Schedule	Actual Duration
Process Plan	Wait Time
Operation	Runtime
Operation	Setup Duration
Operation	Queue Duration

It is recommended that the schedule-related attributes be used exclusively for calendar time duration; therefore, they would match the appropriate start and end dates in the schedule object. Attributes within the process plan and operation objects would be expanded and their definitions clarified.

Each operation would continue to have the runtime, setup and queue attributes. Two attributes would be added to complete the required set of information. A total hours attribute would essentially be the sum of the runtime, setup, and queue times. All of these values will denote manhours or labor without regard for the resources applied to complete the task. The total duration could either be populated by one of the simulation tools or a method could be created in the data model that would sum the values of the individual attributes. If implemented carefully, both options could be made available. In addition, a total duration attribute would be added that defines the duration of the operation while taking into account the resources applied to the task.

The attributes in process plan would be expanded to include values similar to those in the operation. These attributes would essentially be summations of the individual values in the operations that are part of that process plan. In this case it makes sense to implement methods that would sum the individual values if they are not populated by one of the simulation tools.

Given the similarity of the time-related attributes necessary for operation and process plan, it may make sense to create a structure or object with these elements, and use it in each object.

5.2.2 Nested Process Plans

Experience during the SAVE demonstrations revealed that different simulation tools perform their analysis at differing levels of detail in the process plan. As a result, the tools populate data attributes for their operations at the level of their simulation. This may sometimes cause problems when downstream simulation tools need the data, only at a different level of detail.

Although this may not be an explicit data model change, it is important in the successful implementation of the model in either a client or server. When a process plan is highly nested, that is there are operations within the high level plan that are themselves process plans that have their own operations, the data attributes are typically available only at the lowest level. This can cause some problems in use of the data in that relevant information for the higher levels are not available to the simulation tools that need it. For example, resource data is defined for an individual operation. If that operation feeds into another operation in a nested way, the resource data is not available for that higher level operation. The attributes within the operation object that fall into this general category are listed below.

- Precedent Operations
- Personnel Resource Applications
- Tool Resource Applications
- Consumed Parts
- Produced Parts
- Features

The SAVE development team recommends that appropriate logic be developed to define how these items are rolled-up. This logic should be defined in the development specification and implemented at either the client or server level. Server level implementation may be preferred in order to minimize the amount of coding necessary for each client development.

An example for precedent operations is provided here. Figure 5-1 shows a simple nested process plan with precedence defined.

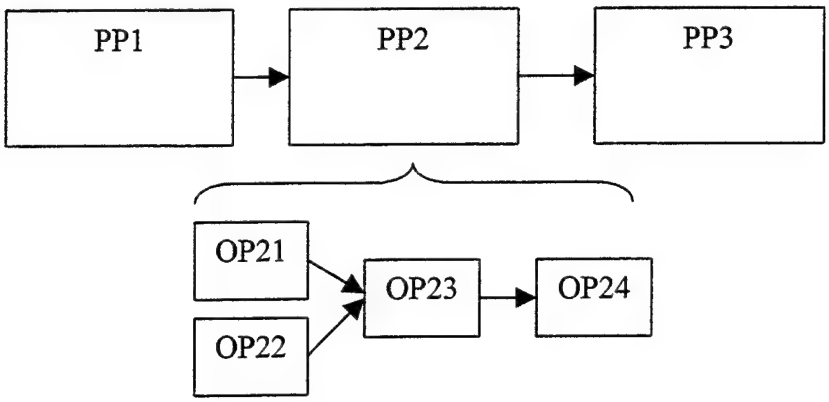

Figure 5-1: Sample Nested Process Plan

In this example, the precedence is straightforward for a tool simulating at the highest level. If a tool needs to simulate the lower level as well, there may be some confusion unless the logic is there to transfer the precedence information. For the lower level, OP21 and OP22 have PP1 as a precedent and the precedent for PP3 changes from PP2 to OP24. This type of logic can be easily built into the code without putting unnecessary burden on the user to define this information explicitly.

Similar logic can be defined for each of the other items, and implementation of the logic will make the model and its use much more robust.

5.2.3 Quantity

Currently, there are quantity attributes related to parts and assemblies in part usage and manufacturing order. Obviously, the quantity in part usage is the quantity of parts either consumed or produced in the operation. The quantity in manufacturing order is the actual number of parts in an order. These two quantities may not be sufficient in performing manufacturing-related simulations.

There are two other quantity-related terms, lot size and batch units, that are quite common in the manufacturing arena. Lot size represents the number of units of a part that are typically produced at one time. The current SDM can handle this if the user chooses to set up the manufacturing orders in lots. This may not, however, always be the reality. Batch units define the quantity of parts that arrive at one time from an external facility. This may not be the same as the number of parts in a particular product.

Future releases of the SDM may need to consider adding information regarding lot size and batch units. Lot size may fit within the manufacturing order as an additional attribute, or within the part (BOM) definition. Batch units will likely fit best in the appropriate part object.

5.2.4 Cost

A general review of the attributes in the cost object is necessary. The objective of the SDM is to provide objects and attributes that are generally applicable in all areas of manufacturing.

Unfortunately, the only real test of the information contained in cost was within the aerospace industry. These attributes should be reviewed by the general industry and modified as appropriate.

5.2.5 Material

Consider adding capacity or quantity attribute. This would expand the material class to include items other than raw material. For example, purchased rivets are bulk material and the capacity would be the quantity.

5.2.6 Tool

Tolerance capability is not necessarily a discrete entity. It may include SPC parameters, design capability, yield, and Cpk. Consider what is needed in this attribute. It may need to be modified so that the tolerance capability is a risk object itself.

5.2.7 Branching Logic

A data sharing scenario was identified during the course of the SAVE demonstration activities that would couple the factory simulation and risk elements of the model. In this scenario, the SDM would contain a decision based routing attribute or structure that would define successor operations based on certain criteria. For example, if an operation fails or has a high probability of failing, the risk attribute would define a different path (either scrap or rework) than if the operation were successful.

Although this may be complicated to define and implement, there are real synergistic benefits to supporting this type of data sharing.

5.2.8 Back Pointers

When traversing nested process plans, the client currently has no way to get back to the parent object. The SAVE team considered building back-pointers into the model to allow both forward and backward traversal. There are strong considerations in this decision, especially in terms of keeping the model purely object oriented. The advantages and disadvantages should be weighed carefully before a decision is made.

5.3 Configuration Management

Some of the elements of configuration management of data within the SDM have not been fully implemented in the "conventional" (C++ / ObjectStore) SDM server. These elements have been fully described in section 2.3.2 of this report and are mentioned here to clearly identify them as areas for future development. Competing approaches to the SAVE Data Model may implement configuration management capabilities differently, but must still provide the basic functionality. In some cases this capability will be inherent in the underlying persistent data store. The following list pertains primarily to the conventional server used for SAVE contract demonstrations.

- Status flags - Flags used to denote the status of data within the SDM are included in the IDL, but are not fully operational within the conventional server. Setting internal flags on all objects and checking their value before updates are allowed must be added to the server code.

- User access control - Currently, there is no mechanism to control access to the SDM on a user or group ID basis. It was decided that this level of control would be implemented largely outside of the CORBA IDL interface. While this is a very important functionality, its implementation will vary with the specifics of each server and the SAVE team made the decision to focus on core server functionality. Client software will have to be responsible for capturing user name and password and passing that information to the server, likely to the DbAccess object which can control access.

- Data object copy capability - This capability is fully designed and is described in section 2.3.2.3 but has not been implemented.

- Distributed back-end data storage - This important element of the SAVE approach to data configuration management has not been implemented in the demonstration server and is discussed in Section 5.4 below.

5.4 Distributed Back-end Data Storage

The primary reason that a CORBA-based architecture was chosen for SAVE was to allow the details of physical data storage to be hidden from the distributed clients. This allows data to be accessed from any number of different physical locations although the data appear to be in the SAVE server. This capability will allow implementation sites to make data from existing databases, including Product Data Management (PDM) systems, transparently available to SAVE-compliant tools. By not forcing replication of data within SAVE, overall data configuration management is made much simpler.

Within CORBA, the action of accessing an attribute within a data object actually executes a method in code, which can perform any number of additional tasks as needed. For example, if a real number value within a data object is actually stored in a relational database (RDB), then the data object internally stores the location of the database and the Structured Query Language (SQL) statements needed to read or update. The client software sees the data going to or coming from SAVE with no knowledge of the underlying RDB transaction.

The concept for distributed back-end data storage is an integral part of the SAVE architecture, but the detailed implementation within the conventional server has not been worked out. Implementation of this feature in other server approaches will be different, but the functionality is the same.

5.5 Use of CORBA Naming Services

A Name Service is a database of names of objects on a particular server. The naming service is used to facilitate the resolution of root names in the architecture to make the architecture more robust to configuration and topology changes. The names are bound to the objects, i.e. a name binding. A name context is an object that contains a set of name bindings in which each name is unique. This is analogous to a directory that contains objects. The names of objects in the name service are resolved from the naming context. A naming service over a distributed network allows a naming context on one machine to make a reference to a name context on another machine. In the case of the SAVE project the naming contexts will be created on each of the hosts and on the common database. The naming context on each host will know the location of the name server on the common database. Therefore the common database will be available to all the hosts.

The SAVE system has not made use of Naming Services to date but does recommend its use in any production system. The primary use will be to simplify the method for clients to locate the SDM and Work Flow servers. In the SAVE demonstrations, the team used hard coded IP addresses, which created an inordinate number of problems even in a small-scale environment. Hard coded addresses fail totally in computer networks that dynamically assign IP addresses. The Work Flow Manager needs a list of user email addresses and as a JAVA application reads these from a file. When the WFM is used as an applet, accessing this information from the naming service is a better solution.

5.6 Performance Improvements

The initial version of the SDM server used for the Interim Demonstration exhibited satisfactory performance (full process plan read/write in approximately 3 minutes) with a realistic 180 operation process plan. However, SAVE beta testing at two sites identified potential performance issues both in client/server network communication speed and in server core memory utilization. Some initial memory leak problems were discovered and quickly resolved, but problems persisted. Beta test process plans had only slightly larger number of operations, but apparently much more information was associated with each operation. By the time the performance issues were clearly identified, it was too late to make significant changes to the version of the SDM server used for the final demonstration.

In the following discussion, bear in mind that some or all of the issues discussed may be peculiar to the specific design of the "conventional" C++ / ObjectStore server used in the SAVE demonstrations and beta tests. Four strongly different ways of developing SAVE-compliant servers have been identified. The "conventional" approach was taken to highlight the fact that SAVE is <u>not</u> dependent on any of the vendor unique approaches which may, in fact, produce better servers. SAVE compliance is defined by adherence to the data and methods specified in the IDL description of the SAVE Data Model. Commercial, competing servers can implement the IDL in any language and in any data persistence mechanism. The IDL abstracts the implementation details from client wrappers, allowing them to be developed independently and used with any compliant server implementation.

Discussions with IONA Corporation, developers of the CORBA ORB used for the current versions of SAVE, have lead to the following conclusions and indications for future development. The CORBA standard is still maturing, and the competitive nature of commercial software leads developers to emphasize extended functionality over robustness and performance. It is generally felt that the steady improvements in hardware speed will provide acceptable performance for distributed object systems. (Keep in mind that the SAVE server to date has only been operated on a desktop PC with 384 MB of memory) This is not unlike the early days of relational databases, and excellent performance was ultimately achieved by a mix of software architecture and hardware speed. The same will be true for CORBA systems in the relatively near future.

Several changes to the SAVE IDL have also been identified that can yield performance improvements. SAVE made virtually everything a first class object, including support for sequences of objects which are used extensively in the model. These decisions were made to minimize the development efforts for each of the client wrappers and to shift the burden for development to the server. Methods could be provided in the server to minimize repetitive software development in multiple client wrappers. Originally there was a belief that many of the objects would require methods in addition to data attributes, indicating first class objects. In fact, relatively few objects have required methods, which opens the door for reconsideration. The downside of these decisions is to increase CORBA network transactions to access these methods and access the highly nested object structure of the model.

Three general approaches to improving performance in the near term have been identified:

1. Port the SDM server to a fast, large memory UNIX platform (the Windows NT platform was only used for economic reasons

2. Add additional methods to some objects to return larger "chunks" of related information in one CORBA network invocation

3. Replace some of the lower-level utility objects with data structures.

Items 2 and 3 are explored below.

5.6.1 Additional Data Access Methods In Some Objects

With the exception of object sequences, all data attributes are accessed one at a time, each requiring a CORBA network transaction. With some consideration, additional methods can be added to key objects to return large structures of data in one CORBA invocation. There are some tradeoffs in this approach because additional server and objectbase time is required to populate these structures if they are not inherent in the object (see item 2 below). It is recommended to add these methods without eliminating the individual attribute access to allow clients to read/update small numbers of attributes when that is all that is required. While the SAVE contract-developed clients tended to access the entire process plan at one time it is expected that mature commercial wrappers will selectively access the data, particularly for update transactions.

Several key data objects could have method added to read and write a structure that contains all data attributes contained in that object. If a client is going to access all of this information anyway, a single CORBA transaction is probably more efficient. These added methods should be considered for:

a) Process Plan

b) Operation

c) Tool

d) Personnel

e) CostInfo

f) ScheduleInfo

g) RiskInfo

h) Part

5.6.2 Increased Use of Data Structures in Objects

The SDM includes a number of low-level support data objects that are used extensively by the higher-level, user-oriented objects. These low-level collections of data were originally made objects to allow related methods to be provided by the server rather than forcing every client to implement the functions that are required on these data. In view of the performance issues (that are certainly aggravated by increased CORBA invocations to access these data and methods) it is recommended to replace these low-level objects with corresponding data attributes, structures, or sequences. In many cases, these objects are used in sequences and the ObjectSequence of objects would be replaced with a CORBA sequence of structures. Some mechanism to share the development of client-side code within the SAVE development community should be considered. Some recommended low-level object replacements include:

- Break - could become a structure in Workshift
- Breakdown - could become a structure in Tool
- Characteristic - could become a structure in all of its using objects
- Contributor - could become a structure in RiskInfo
- CostInfo - could become a structure in all of its using objects
- DateTime - could be made a structure - some allowance for its methods must be made
- Manufacturing Order - could be made a structure in Process Plan

- Part Location - could be made a structure in Part Usage
- Part Usage - could be made a structure in Operation
- Resource Application - could become a structure in Operation
- RiskInfo - could become a structure in all of its using objects
- ScheduleInfo - could become a structure in all of its using objects
- Simulation Model - could become a structure in Process Plan
- Value With Units - could become a structure used in the Characteristics structure
- Versioned Float - could become a structure in all of its using objects - allowance for its methods must be made
- Versioned String - could become a structure in all of its using objects - allowance for its methods must be made
- Work Shift - could be made a structure in Personnel

6.0 Cost Tool Development and Implementation

Four cost models, to be used with Cognition Corporation's Cost Advantage knowledge-based costing tool, were developed under the SAVE contract. The models include:

- Assembly Model
- Sheet Metal Model
- Hand-layup Composites Model
- Machining Model

Copies of these models are available from the Cognition Corporation. This chapter will briefly describe these models and will discuss approaches to tailoring them to new companies and extending their capabilities.

6.1 Introduction

There are several facets to implementing a cost estimating tool into an integrated environment such as SAVE. These include identifying product families, understanding their cost driving features, identifying relevant manufacturing processes, and developing associated cost estimating relationships. The developers also need to work closely with their ultimate system users and data sources to ensure the best models and end-user buy-in for the system.

The first step towards implementing the SAVE cost estimating system is to work with the cost estimators, designers and manufacturing personnel to identify the components that are most beneficial to include in the system. Next, identify the cost driving features of these part families and relate them to the relevant manufacturing processes. In-depth research is then required to define manufacturing planning performed at the factory, limitations of the equipment, material specifications, time standards, and cost factors.

Resources for performing this development task include:

- Manufacturing Engineers
- Process Experts
- Producibility Engineers
- Textbooks and Handbooks
- Industrial Engineers
- Value Engineers or Cost Estimators
- Finance Personnel

- Tool Designers

Once the research is complete, the next phase is design. This encompasses the establishment of variables and the designation of variable location within the cost tool. Relationships to a SAVE compliant database are also established here. The next phase is to program the variables and cost estimating relationships (CERs) into the cost tool utilizing templates like those developed under SAVE. Once this phase is complete, a validation activity is required to make sure the information is reliable. It is important to include the end users in these activities so that they will be comfortable with the features and CER approaches that are selected.

Cost Advantage™ contains three variable categories: material, process, and feature. The cost and design characteristics are allocated into these three areas. A typical developer's screen is shown below in Figure 6-1. Cost estimators or value engineers are typically the ones who will be implementing the cost estimating relationships into this tool. A producibility engineer or manufacturing engineer will provide producibility rule support. It is critical to document the model with comments about the CERs and producibility guidance. Cost Advantage™ provides the capability for the developer to record internal notes regarding each object or formula. Other help information can be documented for access by the end users. This help can be embedded in the cost tool, located in external files, or accessed from the web. The user can easily access this information while working on the cost trade.

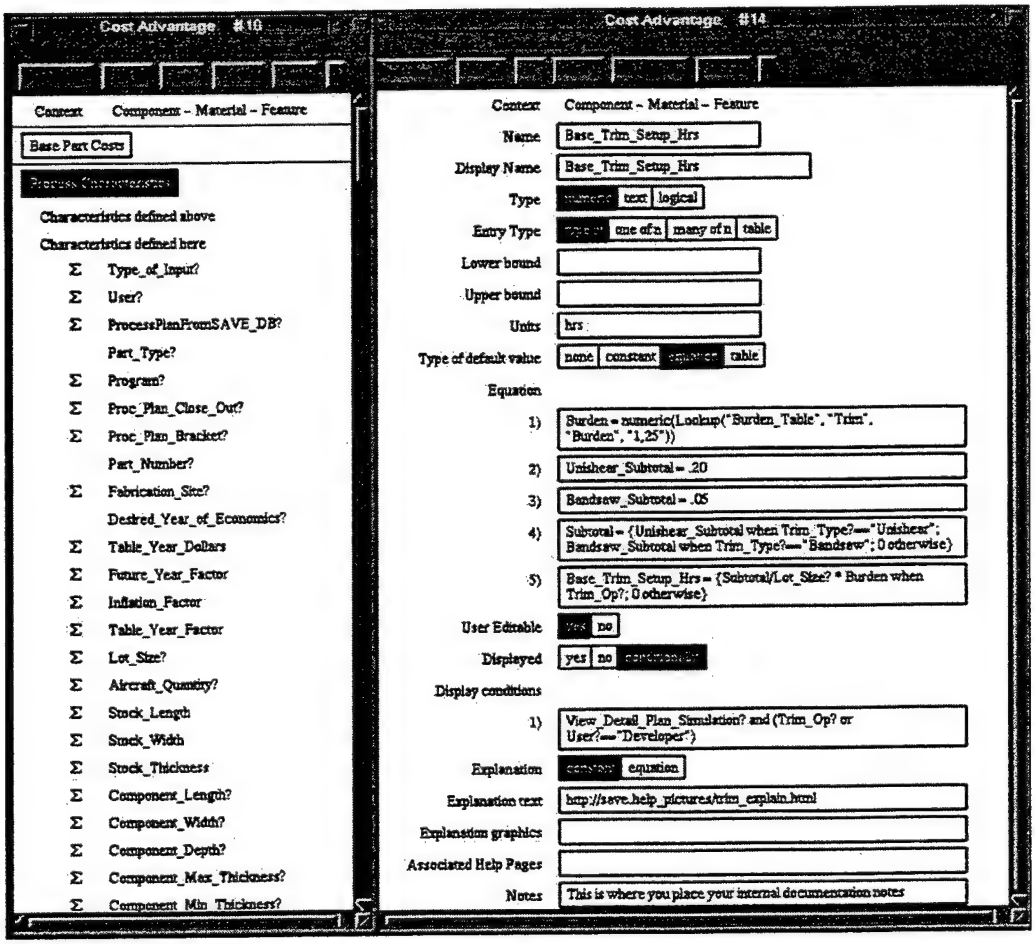

Figure 6-1. Cost Advantage™ Development Environment Screen

Both cultural and political issues need to be considered when implementing an expert system cost model such as the SAVE tool. Agreement is required by all affected departments for this tool to be accepted and utilized. This is a new way to do business for many companies, so this acceptance is critical to the success of the program. This cost tool provides the Integrated Product Team (IPT) a way to rapidly do design trades that include cost. The designer could potentially use this tool on his own, although this should only occur for straightforward trades. The bounds for a designer using this tool without a cost estimator need to be understood and agreed to by all groups. The ideal situation for using this tool is for the designer and cost estimator to sit together and utilize the SAVE cost tool during their design trade.

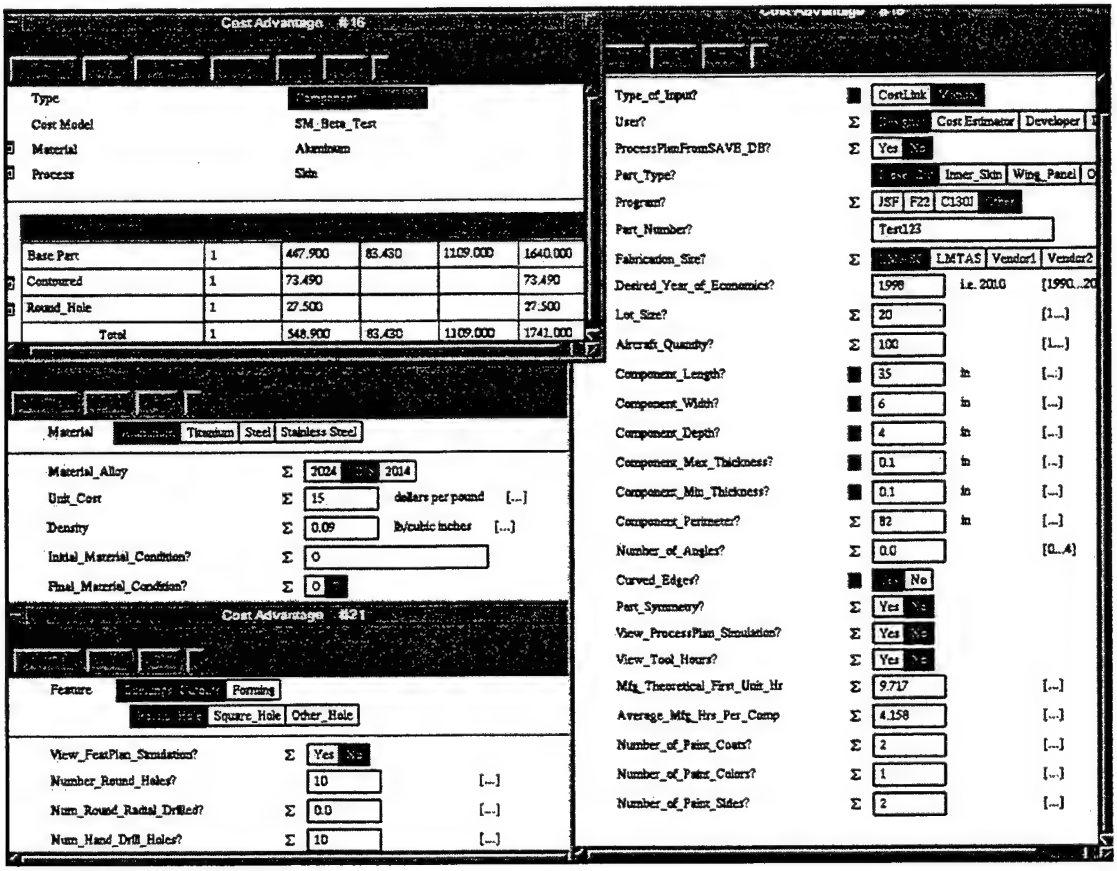

Figure 6-2. End User Screen Example

An example of the type of information that the end user would see when utilizing the SAVE cost tool is shown in Figure 6-2. Both inputs and outputs are readily accessible during the trade study. The user can also query the system for help during his session. When implementing the system, the developer should work with the end users to ensure that the appropriate information is presented on the user's screen.

There are several things that can be done to maximize the benefit and usefulness of the system. First, training is very important for both the users and developers. Secondly, system

maintenance is required to avoid the potential problem of data obsolescence. Developing a plan for updating the CER's when the factory and products evolve can do this. This plan should include a scheme for material costs and labor rate updates.

6.2 SAVE Cost Model Conventions and Model Updating Methods

This section describes modifications to the cost models for a particular plant application. The extensive notes and explanations in the models will be quite useful when upgrading the models in the future.

6.2.1 Assembly Cost Model

The assembly model is developed to work directly with the SAVE system. It requires that a process plan is available to rapidly obtain a cost estimate. Each manufacturing operation, such as locate, setup, assemble, or drill, is an individual Cost Advantage™ "Feature". This allows the data to flow directly from SAVE in an efficient manner. This is different from the component models where "Feature" means a design feature, not a manufacturing operation. The features currently in the assembly model include:

Setup	Align	Locate
Drill / Drill Ream	Resistance Spotweld	Back Drill
Bench Drill	Spot Face	Drill Out
Ream	Finish Ream	Seal
Verify	Record	Torque
Inspect	Install	Assemble
Attach	Remove	Shim
Cold Work	Packing	Bond Check
Deburr	Apply	Rivet

6.2.1.1 Modify Cost Estimating Relationships for an Existing Operation

The CER for a manufacturing operation is placed in the variable titled that operation; i.e., the formulas for performing a locate operation are in the variable named "locate". These variables can be modified to reflect a company's formulas and capabilities.

6.2.1.2 Adding a New Operation

Identify the cost drivers for the new operation. Add these design features, and create a new variable for the CER formula. Follow the naming and display conventions from an existing operation.

6.2.1.3 Modifying Cost Factors and Labor Rates

Labor rates and factors are located in a set of external tables. These are ASCII text files that are updated with the most recent factory data.

6.2.1.4 Modifying Top Level Cost Relationships

Cost relationships, such as non-recurring labor or quality assurance labor costs, can be modified or new ones added to reflect the company's business practices. If new ones are added, make sure that they are included in the total roll-up.

6.2.2 Sheet Metal Cost Model

The developer should identify the key cost driving features for the company's manufacturing processes. Figure 6-3 shows representative features for use in a sheet metal cost model. Cost estimating relationships are developed for these features and entered into the cost-estimating model. To conform to Cost Advantage™ conventions, base costs and feature costs are split. For example, a contour is a feature because it is achieved by doing secondary processing on the part. For the sheet metal component, only the basic layout and shearing processes are included as part of the base and material costs.

Modification of cost factors, material cost information, material data, and top level cost relationships is handled similarly to the Assembly model in external ASCII files.

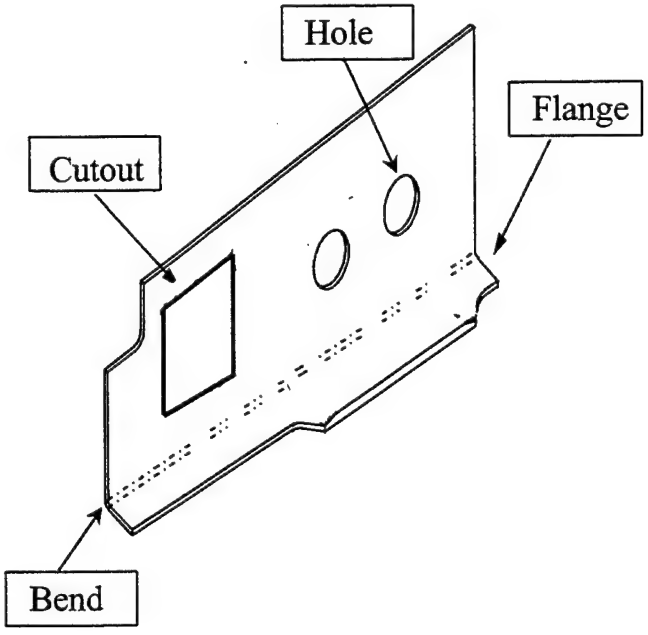

Figure 6-3: Sample Sheet Metal Design Features

The following manufacturing operations are currently in the Sheet Metal Cost Model:

Layout	Shear	Drill
Rout	Hydroform Fluid Cell or Hydraulic Press	Corrosion Protection
Heat Treat	Age Harden	Inspect
Mark	Sand	Trim
Mask	Clean	Deburr
Straighten		

The model is designed around providing the capability to add additional part families as well as additional manufacturing operations, cost estimating relationships, and design features. The Sheet Metal Model currently includes the following design features:

- Openings and Cutouts
  - Round Holes
  - Square Holes
  - Other Holes
- Forming
  - Bends
  - Joggles
  - Flanges
  - Beads
- Contour
- Material
  - Type and Alloy
  - Density
  - Initial and final material condition

6.2.3 Machining Cost Model

The developer should identify the key cost driving features for the company's manufacturing processes. Figure 6-4 shows representative features for use in a machining cost model. Cost

estimating relationships (CERs) are developed for these features and entered into the cost-estimating model. The formulas for an operation are not located in the Cost Advantage™ model. They are processed externally to Cost Advantage™ in a computer-generated spreadsheet (i.e., Microsoft® Excel) to create a set of factors for different constraints. These are located in a table in the same directory as the cost model. The composites cost model follows this same convention. This provides a second example on how to lay out a model. To conform to Cost Advantage™ conventions, base costs and feature costs are split.

Modification of cost factors, material cost information, material data, and top level cost relationships is handled similarly to the Assembly model in external ASCII files.

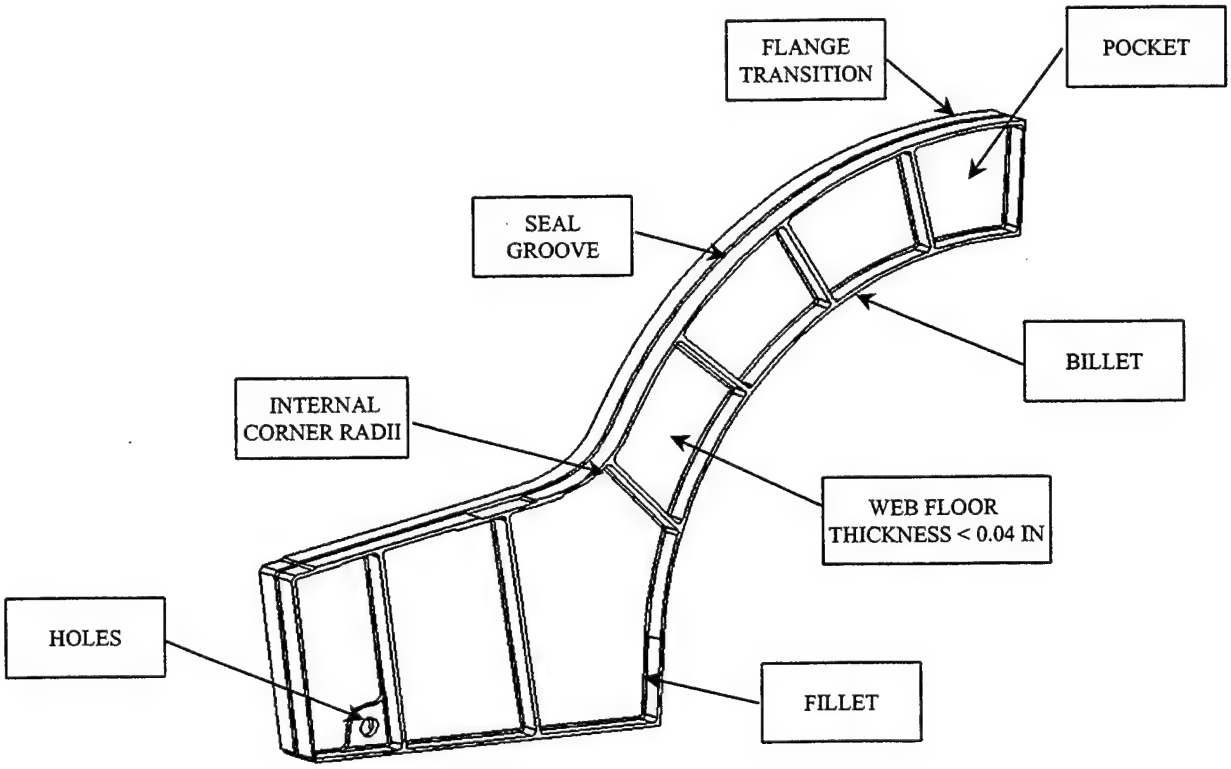

Figure 6-4: Sample Machining Features

Additional features that are sensitive to cost also can be added along with their associated cost estimating relationships. The current version of the Machining Model includes these features:

- Pocket
- Holes
  - Cut Out
  - Fastener hole

- End cut or angle cut
- Material
  - Type
  - Billet Thickness
  - Temper
  - Product form

6.2.4 Composites Cost Model

The developer should identify the key cost driving features for the company's manufacturing processes. Figure 6-5 shows representative features for use in a composites cost model. Cost estimating relationships (CERs) are developed for these features and entered into the cost-estimating model. To conform to Cost Advantage™ conventions, base costs and feature costs are split.

Modification of cost factors, material cost information, material data, and top level cost relationships is handled similarly to the Assembly model in external ASCII files.

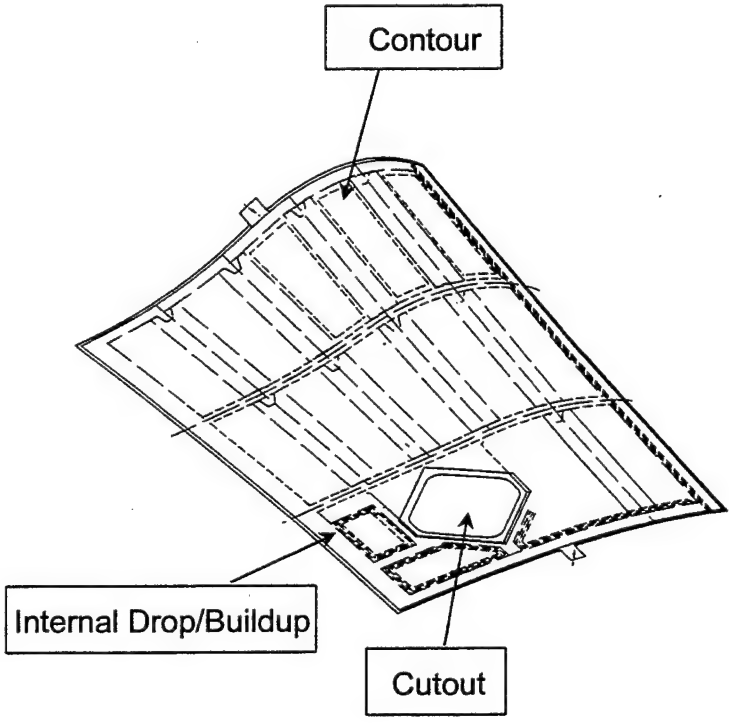

Figure 6-5: Sample Composite Part Characteristics

Additional features that are sensitive to cost also can be added along with their associated cost estimating relationships. The manufacturing, materials, and process team develop these features and their associated cost estimating relationships with support from the cost estimating organization. The current version of the Composites Model includes these feature categories:

- Contour
- Internal Drop / Buildup
- Cutout
- Plies
- Material
  - Type and Form

6.3 Recommended Enhancements

The following are some lessons learned and suggestions for future model revisions. The Cost Advantage™ format allows for quite extensive model development and customization.

- Add additional activity-based costing (ABC) algorithms.
- Enhance non-recurring and tooling algorithms to meet the company's practices.
- Add percent derivative function to apply a learning curve to only part of the component cost.
- Add more producibility guidance and web-based help to match the company's capabilities.
- With user guidance, assure consistent displays and order of information
- For the case where the designer is getting an initial or intermediate cost estimate before the component design is completed, add rules or CERs into the cost model that account for design features that may not have been added at the point the estimate is made. Also, add a warning note that the cost estimate is incomplete and for reference only if there is not a complete description of the component.

APPENDIX A - SAVE Deliverable Software Listing

The SAVE-developed software that is described in this report has been delivered on CDROM to the U.S. Air Force Research Laboratory. Requests for copies of this software should be directed to Mr. James W. Poindexter, SAVE Program Manager (james.poindexter@afrl.af.mil). The files on the CDROM include software source code, make files, and documentation. An outline of the directory structure included follows.

Subdirectories are:

1. Server: This directory contains all the C++, Orbix-related, and Objectstore-related source code for running and editing the clients and server that implement the save.idl.

2. Java: This directory contains all the Java source code for running the work flow manager, the query manager, and a sample simulator tester that implement the simulator.idl.

The JAVA directory is further subdivided as follows:

```
|-SAVE
    | |-batch   (Batch files to run the QM, WFM and tester and jarred class files)
    | |-idl     (Th IDL files)
    | |-src     (Surce code for WFM, QM and simulation tester)
    | |-example (Source code for example simulators)
    | |-classes (WFM, QM, tester and example simulator classes)
    | +-doc     (HTML documentation)
+-README
```

APPENDIX B - SAVE Vendor Product Descriptions

DENEB QUEST - Deneb's QUEST is a 3D physically-based simulation tool used to analyze production scenarios, product mixes and failure responses for machines and labor; factory layout; throughput; and production costs. QUEST uses a physically motivated approach to simulation modeling. Each resource is intuitively described as it exists in the real world. These resources are then visually connected to describe the flow of parts or information. Built-in logic options are written in a high-level procedural language that is provided for rapid model creation. All of these routing and processing options are available for user modification. Full scale, 3D geometry can be imported through a wide range of CAD translators or created in the integrated CAD system provided with QUEST. Models can be executed with or without animation and changes can be made interactively during a run session. Models may also be executed in a batch mode. Access is provided to a full range of analysis features including bottleneck identification, confidence interval calculations, optimization algorithms, and custom summaries for resource allocation and system performance.

DENEB ERGO - Deneb's ERGO enables the user to rapidly prototype human motion within a workplace, then perform ergonomic analysis on the designed job(s). The following functions and tools provide the ability to accomplish these tasks:

- A dedicated human motion programming interface that includes inverse kinematics and graphical programming for the human models.
- 50, and 95 percentile male and female models.
- An energy expenditure prediction model.
- Guidelines for two-handed lifting.
- A posture analysis system.
- A work-measurement standard to estimate time standards for jobs.

The ERGO tool allows the anthropometry human factors specialist to evaluate if a particular worker can perform the required task with regard to "reachability." With Deneb/ERGO, the human factors specialist can determine if a worker's anthropometry will allow him to work comfortably in an already existing workplace, then determine the percentage of the workforce that could comfortably perform the same or similar task in the same or similar workspace.

Deneb/ERGO enables the industrial engineer to ensure that the workplace design allows workers to complete their job in the allotted time. In addition, the industrial engineer can develop time standards for new jobs and improve time standards for existing tasks by using this tool.

The industrial ergonomist can use the Deneb/ERGO tool to evaluate the lifting capacity of workers. In addition, the ergonomist can determine the percentage of the workforce that has the physical capacity to do a particular job and interactively examine the working postures that

might endanger the health of the workers. For high exertion tasks, the energy expenditure requirements can be obtained to determine job rotation requirements.

Once the initial analysis is complete, the model is used as a teaching tool to ensure workers correctly perform their job. ERGO works in conjunction with the IGRIP 3D environment.

DENEB IGRIP - Deneb's IGRIP provides a physics-based, 3D environment specifically for designing, verifying and rapid prototyping of concept designs involving structures, mechanical systems and humans.

IGRIP can visualize and evaluate concept designs for complex systems and subsystems. The CAD-based models represent the actual geometry and motion characteristics associated with the real world system. Incorporating dynamics into the physics-based models allows the user to create an environment that supports simulation based design. In this environment, users are able to design, build, test, operate and support multiple product and system scenarios in a fast, efficient and cost-effective manner. Examples include: Complex mechanical designs; rigid body dynamic analysis, mobile systems/vehicles, satellite design and deployment, design verification, concurrent engineering, human factors engineering (w/ERGO option), mission planning and space task engineering. With its robotic capabilities, IGRIP is also a multi-purpose engineering tool for designing, evaluating and off-line programming of robotic workcells.

SAIC ASURE - The Analytical System for Uncertainty and Risk Estimation (ASURE), a virtual prototyping (VP) tool, supports a method for better decision making in any development and/or acquisition process. ASURE assists decision makers in incorporating risk and uncertainty in system or concept evaluation decisions based on limited test data and science-based models for estimating system characteristics. The ASURE tool

- Permits the capture and management of evolving system concept descriptions;
- Provides easy access to available data for effective analysis; and
- Allows rigorous uncertainty management.

ASURE has three major components with the following principle functions:

- The graphical user interface: Capture user input and display system descriptions and results.
- The information base: Represent and maintain system descriptions using form & function decomposition and process/environment models.
- The computational kernel: Compute input distributions, perform uncertainty propagation, and compute uncertainty in outputs. This piece is also referred to as PIE, the Probabilistic Inference Engine.

The ASURE function concept is represented in Figure B-1.

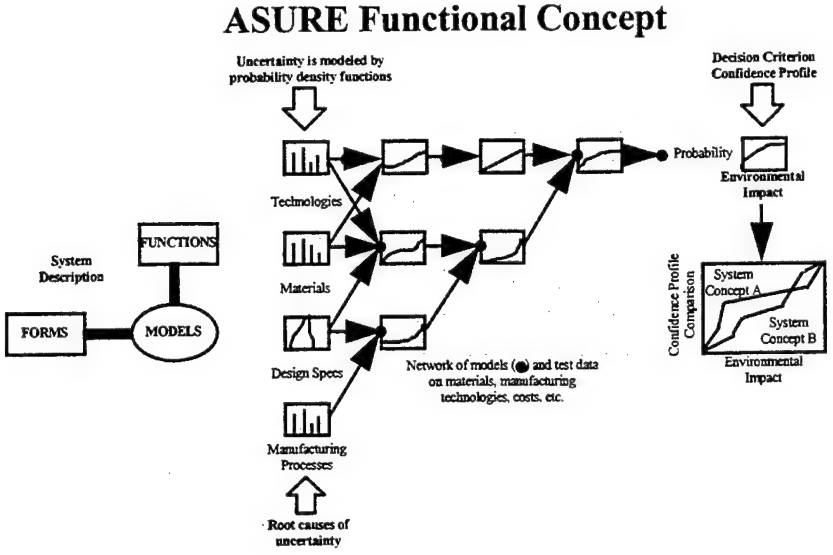

Figure B-1 ASURE Functional Concept

(ref: SAIC's ASURE JAST-SAVE Training Presentation, Jan. 4-5, P. 7)

SYMIX FACTOR/AIM - Symix's FACTOR/AIM is an integrated application within the FACTOR Finite Capacity Management System. FACTOR/AIM is used by industrial and manufacturing engineers to efficiently and effectively design and improve manufacturing operations. It supports the Total Capacity Management activities of Capacity Design and Continuous Capacity Improvement. AIM models can be used by other FACTOR scheduling applications that provide support for capacity, logistics & production scheduling, and schedule management activities.

AIM is built around a relational database that stores the manufacturing operation description and simulation output. The description of the manufacturing process used by AIM can be created through the AIM user interface or by using existing manufacturing enterprise data. Part descriptions, process plans, order release schedules, machine locations and descriptions, shift schedules, etc. can be transferred from other data sources into the AIM database, for use as part of the model. Using existing data increases both modeler productivity and model accuracy.

FACTOR/AIM Scheduling improves the utilization of manufacturing resources and the ability of the management staff to accurately maintain and communicate resource assignments and schedules. FACTOR/AIM Scheduling is also used to predict future resource requirements based upon projected volumes. The production area status and new orders are entered into the system via spreadsheets. Projected schedules are then created in textual form for distribution, and in graphical GANTT chart form adjustment and execution.

FACTOR/AIM component relationships are depicted in Figure B-2.

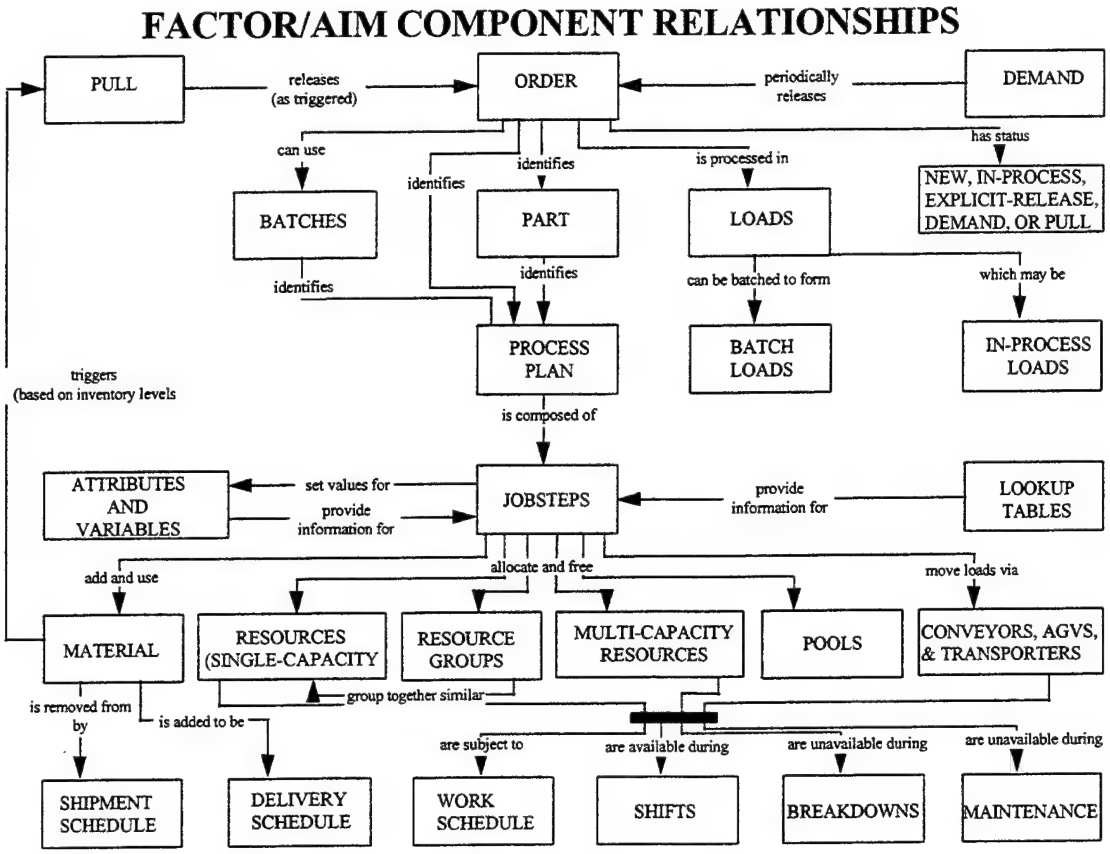

Figure B-2 FACTOR/AIM Component Relationships

COGNITION COST ADVANTAGE - Cost Advantage (CA) is a Design for Manufacturability (DFM) expert system that provides immediate cost data, design guidance, and producibility analysis. It captures manufacturing process knowledge and leverages that information to identify costs drivers through all stages of a product's life cycle.

Cost Advantage can be keyboard driven, can accept part geometry directly from feature-based solid modelers or receive parameters from an external source so that cost and producibility analysis occurs during the design cycle. Through available application linking facilities that bind Cost Advantage directly to the solid modeling session, accurate and immediate feedback is provided to the engineer while geometry construction is underway. Analysis against the design and manufacturing rules of your corporation provides design guidance that ensures identification of high cost drivers as they are introduced, and provides ready alternatives that can reduce costs and increase product reliability. The process/data flow for Cost Advantage use is shown in Figure B-3. An Integrated Product Process Database (IPPDB) houses the information from which CA can draw information such as Cost Estimating Relationships (CER) documentation and legacy rates & factors.

PROCESS/DATA FLOW FOR COST ADVANTAGE USE

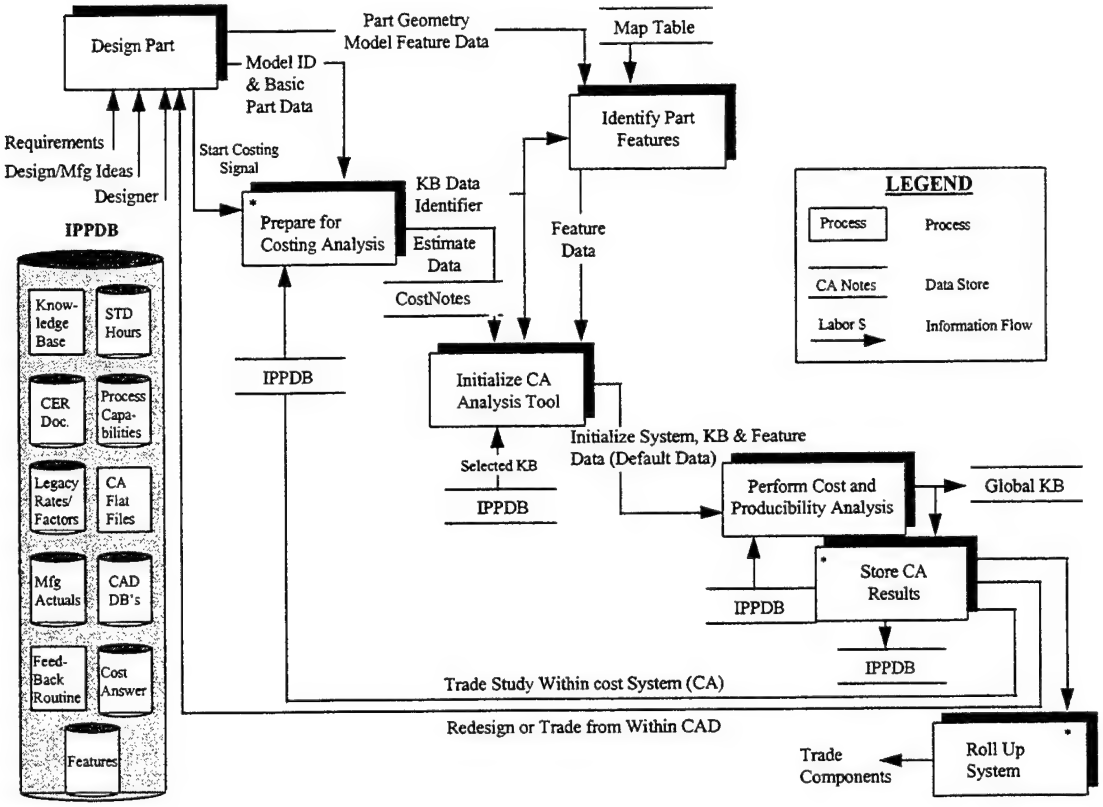

Figure B-3 Process/Data Flow For Use Of Cost Advantage

VSA - These tools predict and optimize manufacturing and assembly variation before tooling is committed. VSA enables the engineer to optimize design tolerance and manufacturing variations to achieve the highest quality at the most cost effective price. VSA-Builder is a graphics pre-processor which accepts part geometry from most CAD systems to interactively create the VSA model. VSA promotes structured communication between design and manufacturing, predicts assembly variation and/or product quality (functional variations) before component parts are manufactured and eliminates design and process incompatibilities.

VSA identifies the location and percent of contribution of each critical variation in the assembly, closes the dimensional analysis loop by interfacing directly with Geometric Dimensioning & Tolerancing (GD&T), design of experiments and statistical process control data.

VSA's focus has been on developing the software tools and engineering expertise to support the implementation of a structured Dimensional Management Process in large engineering, manufacturing and product assembly organizations. The Dimensional Management Process centers on integrating six key ingredients for dimensional quality into the product release cycle through well-defined deliverables:

1. All product dimensional requirements are clearly defined at the beginning of the design cycle.

2. The design, manufacturing and assembly process as specified optimally meets product requirements.

3. Product documentation is correct and communicated effectively throughout the assembly methods and process.

4. A measurement plan is implemented that validates product requirements.

5. Manufacturing capabilities achieve design intent.

6. A well-defined production to design "feedback loop" exists.

Figure B-4 shows the Dimensional Management Process employed by VSA.

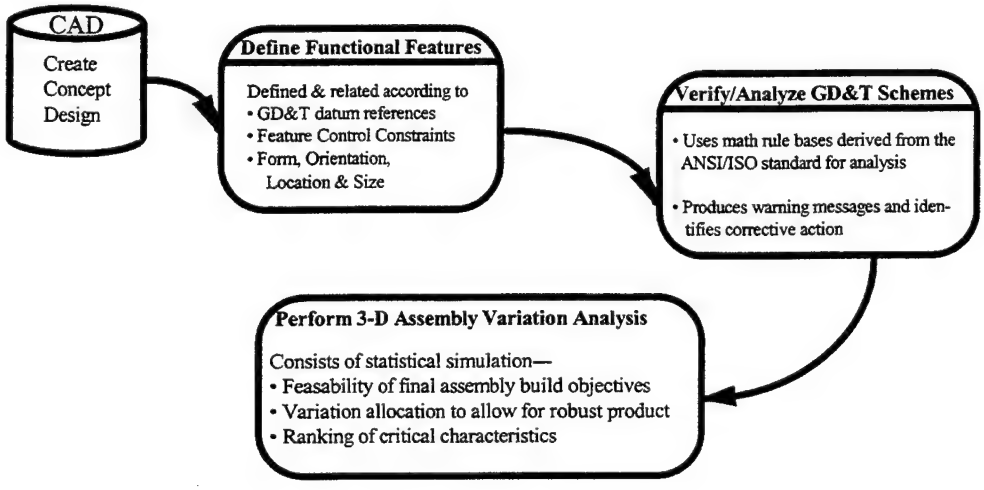

Figure B-4 VSA Dimensional Management Process Tool Capabilities

APPENDIX C - Tool Specific Input/Output Capabilities

The table shown below identifies the how the simulation tools currently integrated into SAVE utilize the Process Plan within the SAVE Data Model. The fully expanded nesting of data within a Process Plan is shown, and each data field notes which tools interact with that field as Input (I), Output(O), or both (I/O). This table is current as of the SAVE Final Demonstration.

This matrix is an excellent planning aid for developing new tool interfaces, as it clearly shows which data can be provided by other tools, and what data other tools expect to be able to use as input.

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Process Plan						
Average Time Critical Path		I/O				
Date Time						I
Description	I/O		I/O	I/O	I/O	I/O
Name		I/O	I/O	I/O	I/O	I/O
Status	I	I	I	I	I	I
Waiting Time		I/O	O		I	
Process Plan/Characteristics						
Date Time						I
Description						I
Name		I/O				I
Numerical Value						I
Textual Value		I/O				I
Process Plan/Cost						
Average Production Unit Cost		O				
Base Year		I				
Development Tooling Cost		O				
Fiscal Year		O				
Labor Inflation Factor		I				
Material Inflation Factor		I				
Material Cost		O				
Non-Recurring Tooling Cost		O				
Other Non-Recurring Manufacturing Cost		O				
Other Recurring Manufacturing Cost		O				
Plant Equipment Cost		O				
Production Tooling Cost		O				
Quality Assurance Cost		O				
Recurring Manufacturing Labor Cost		O				
Sustaining Tooling Cost		O				
Tool Material Cost		O				

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Sustaining Tooling Cost		O				
Tool Material Cost		O				
Total Manufacturing Cost		O				
Type		O				
Inflation Table						
Process Plan/Manufacturing Order						
Date Time						
Description						
Name						
Quantity						
Process Plan/Operation						
Critical Path Step		I/O				
Date Time						
Description			I/O	I/O	I/O	I
ID		I/O	I/O	I/O	I/O	I
Name	I/O	I/O	I/O	I/O	I/O	I
Precedent Operations	I/O		I/O		I/O	I
Quantity	I/O	I/O		I/O	I/O	I
Queue Avg Capacity						
Queue Duration						
Queue Total Capacity						
Run Time		I/O	I/O	O	I/O	
Setup Description						
Setup Duration		I		I/O	I/O	
Type		I/O				I
Process Plan/Operation/ Characteristics						
Date Time		I/O				I
Description						I
Name	I/O	I/O				I
Textual Value	I/O	I/O				I
Numerical Value	I/O					I
Process Plan/Operation/Cost						
Average Production Unit Cost		O				
Base Year		I				
Development Tooling Cost		O				
Fiscal Year		O				

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Labor Inflation Factor		I				
Material Inflation Factor		I				
Material Cost		O				
Non-Recurring Tooling Cost		O				
Other Non-Recurring Manufacturing Cost		O				
Other Recurring Manufacturing Cost		O				
Plant Equipment Cost		O				
Production Tooling Cost		O				
Quality Assurance Cost		O				
Recurring Manufacturing Labor Cost		O				
Sustaining Tooling Cost		O				
Tool Material Cost		O				
Total Manufacturing Cost		O				
Type		O				
Inflation Table						
Process Plan/Operation/ Features						
Date Time						
Description						I
Name	I/O					I
Quantity	I/O					I
Type						I
Process Plan/Operation/Features/ Characteristics						
Date Time						
Description						
Name	I/O					
Textual Value	I/O					
Numerical Value	I/O					
Process Plan/Operation/Features/Cost						
Average Production Unit Cost						
Base Year						
Development Tooling Cost						
Fiscal Year						
Labor Inflation Factor						

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Material Inflation Factor						
Material Cost						
Non-Recurring Tooling Cost						
Other Non-Recurring Manufacturing Cost						
Other Recurring Manufacturing Cost						
Plant Equipment Cost						
Production Tooling Cost						
Quality Assurance Cost						
Recurring Manufacturing Labor Cost						
Sustaining Tooling Cost						
Tool Material Cost						
Total Manufacturing Cost						
Type						
Inflation Table						
Process Plan/Operation/Part						
Associated Parts				O		
Complexity						
Date Time						
Description				I/O	I/O	
Family				I/O	I/O	
Name	I/O			I/O	I/O	
Number				I/O		
Quantity	I/O				I/O	
Rejection Rate	I/O					
Type						
Process Plan/Operation/Part/CAD Model						
Date Time						
Description				I/O	I/O	
Envelope X, Y, Z						
Format				O	O	
Location				I/O	I/O	
Name				I/O		
Process Plan/Operation/Part/Cost						
Average Production Unit Cost						

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Base Year						
Development Tooling Cost						
Fiscal Year						
Labor Inflation Factor						
Material Inflation Factor						
Material Cost						
Non-Recurring Tooling Cost						
Other Non-Recurring Manufacturing Cost						
Other Recurring Manufacturing Cost						
Plant Equipment Cost						
Production Tooling Cost						
Quality Assurance Cost						
Recurring Manufacturing Labor Cost						
Sustaining Tooling Cost						
Tool Material Cost						
Total Manufacturing Cost						
Type						
Inflation Table						
Process Plan/Operation/Part/Feature						
Date Time						
Description						
Name	I/O					
Quantity	I/O					
Type						
Process Plan/Operation/Part/Feature/ Characteristics						
Date Time						
Description						
Name	I/O					
Textual Value	I/O					
Numerical Value	I/O					
Process Plan/Operation/Part/Feature/Cost						
Average Production Unit Cost						
Base Year						

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Development Tooling Cost						
Fiscal Year						
Labor Inflation Factor						
Material Inflation Factor						
Material Cost						
Non-Recurring Tooling Cost						
Other Non-Recurring Manufacturing Cost						
Other Recurring Manufacturing Cost						
Plant Equipment Cost						
Production Tooling Cost						
Quality Assurance Cost						
Recurring Manufacturing Labor Cost						
Sustaining Tooling Cost						
Tool Material Cost						
Total Manufacturing Cost						
Type						
Inflation Table						
Process Plan/Operation/Part/Material						
Date Time						
Description						
Form						
Name						
Type						
Unit Cost						
Process Plan/Operation/Part/Material/ Characteristics						
Date Time						
Description						
Name						
Textual Value						
Numerical Value						
Process Plan/Operation/ Process Plan Refer Back to Top of Page						

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Process Plan/Operation/ Reference Process						
Complexity	I/O					
Date Time						
Description						
Maturity	I/O					
Name	I/O					
Operation Characteristics						
Stability						
Standard Hours						
Process Plan/Operation/ Reference Process/ Characteristics						
Date Time						
Description						
Name	I/O					
Textual Value	I/O					
Numerical Value	I/O					
Process Plan/Operation/ Reference Process/Risk						
Consequence of Failure						
Cp						I/O
Cpk						I/O
Description						I/O
Mean	I/O					I/O
Probability of Failure	O					I/O
Qualitative Results						I/O
Standard Deviation						I/O
Yield	O					I/O
Process Plan/Operation/ Reference Process/Risk/ Contributors						
Date Time						I/O
Description						I/O
Name	I/O					I/O
Percent Contribution	I/O					I/O
Process Plan/Operation/ Resource Application						
Date Time						
Description						
Name						
Transformation Matrix			I/O	I/O	I/O	
Quantity					I/O	
Process Plan/Operation/ Resource Application/ Resource						
Date Time						
Description			I/O	I/O	I/O	
Name			I/O	I/O	I/O	
Efficiency						

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Process Plan/Operation/ Resource Application/ Resource/CAD Model						
Date Time						
Description						
Envelope X, Y, Z						
Format				O	O	
Location				I/O	I/O	
Name						
Process Plan/Operation/ Resource Application/ Resource Pool						
Date Time						
Description			I/O	I/O	I/O	
Name			I/O	I/O	I/O	
Quantity			I/O	I/O	I/O	
Utilization					O	
Process Plan/Operation/ Resource Application/ Resource Pool/Resource						
Date Time						
Description		I/O	I/O	I/O	I/O	
Name			I/O	I/O	I/O	
Efficiency		I		I/O		
Process Plan/Operation/ Resource Application/ Resource Pool/Resource/CAD Model						
Date Time						
Description						
Envelope X, Y, Z						
Format				O	O	
Location				I/O	I/O	
Name						
Process Plan/Operation/Risk						
Consequence of Failure						I/O
Cp						I/O
Cpk						I/O
Description						I/O
Mean	I/O					I/O
Probability of Failure	O					I/O
Qualitative Results						I/O
Standard Deviation						I/O
Yield	O					I/O
Process Plan/Operation/Risk/Contributors						
Date Time						I/O
Description						I/O
Name	I/O					I/O
Percent Contribution	I/O					I/O

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Process Plan/Operation/ Schedule						
Actual Duration				O	O	
Actual End Date						
Actual Start Date						
Planned Duration						
Planned End Date						
Planned Start Date						
Priority						
Process Plan/Part						
Associated Parts						
Complexity						
Date Time						
Description						
Family		O				
Name	I/O	I/O				
Number		O				
Quantity	I/O					
Rejection Rate	I/O					
Type		I/O				
Process Plan/Part/ CAD Model						
Date Time						
Description						
Envelope X, Y, Z						
Format						
Location						
Name						
Process Plan/Part/ Cost						
Average Production Unit Cost		O				
Base Year		I				
Development Tooling Cost		O				
Fiscal Year		O				
Labor Inflation Factor		I				
Material Inflation Factor		I				
Material Cost		O				
Non-Recurring Tooling Cost		O				
Other Non-Recurring Manufacturing Cost		O				
Other Recurring Manufacturing Cost		O				
Plant Equipment Cost		O				

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Production Tooling Cost		O				
Quality Assurance Cost		O				
Recurring Manufacturing Labor Cost		O				
Sustaining Tooling Cost		O				
Tool Material Cost		O				
Total Manufacturing Cost		O				
Type		O				
Inflation Table						
Process Plan/Part/ Feature						
Date Time						
Description						
Name	I/O					
Quantity	I/O					
Type						
Process Plan/Part/ Feature/ Characteristics						
Date Time						
Description						
Name	I/O					
Textual Value	I/O					
Numerical Value	I/O					
Process Plan/Part/ Feature/Cost						
Average Production Unit Cost						
Base Year						
Development Tooling Cost						
Fiscal Year						
Labor Inflation Factor						
Material Inflation Factor						
Material Cost						
Non-Recurring Tooling Cost						
Other Non-Recurring Manufacturing Cost						
Other Recurring Manufacturing Cost						
Plant Equipment Cost						
Production Tooling Cost						

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Quality Assurance Cost						
Recurring Manufacturing Labor Cost						
Sustaining Tooling Cost						
Tool Material Cost						
Total Manufacturing Cost						
Type						
Inflation Table						
Process Plan/Part/ Material						
Date Time						
Description						
Form						
Name		I				
Type		O				
Unit Cost						
Process Plan/Part/ Material/ Characteristics						
Date Time						
Description						
Name						
Textual Value						
Numerical Value						
Process Plan/Resource Pool						
Date Time						
Description			I/O	I/O	I/O	
Name			I/O	I/O	I/O	
Quantity			I/O	I/O	I/O	
Utilization					O	
Process Plan/Resource Pool/ Resource						
Date Time						
Description			I/O	I/O	I/O	
Name			I/O	I/O	I/O	
Efficiency		I/O				
Process Plan/Resource Pool/ Resource/CAD Model						
Date Time						
Description						
Envelope X, Y, Z						
Format				O	O	
Location				I/O	I/O	
Name						

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Process Plan/Risk						
Consequence of Failure						
Cp						
Cpk						
Description						
Mean	I/O					
Probability of Failure	O					
Qualitative Results						
Standard Deviation						
Yield	O					
Process Plan/Risk/ Contributors						
Date Time						
Description						
Name	I/O					
Percent Contribution	I/O					
Process Plan/Schedule						
Actual Duration			O			
Actual End Date			O			
Actual Start Date			O			
Planned Duration						
Planned End Date						
Planned Start Date						
Priority						
Process Plan/Simulation Model						
Data Location				O	O	O
Date Time				O	O	O
Description				O	O	O
Factory Model						
Name				O	O	O
Simulation Code				O	O	O
Type				O	O	O

Resource Subclass Relationship

Applies to Each Instance of Resource Unless Otherwise Noted.

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Resource						
Date Time						
Description			I/O	I/O	I/O	
Name			I/O	I/O	I/O	
Efficiency						
Resource/Personnel						
Skill				I/O		
Labor Rate						
Labor Rate Year						
Resource/Personnel/ Work Calendar						
Date Time						
Description						
Name					I/O	
Number of Work Days					I/O	
Work Year						
January 1 Day of Week						
Resource/Personnel/Work Shift						
Date Time						
Description						
End Time					I/O	
Hours in Shift						
Name					I/O	
Start Time					I/O	
Resource/Personnel/Work Shift/Break						
Start Time					I/O	
End Time					I/O	
Resource/Tools						
Cost						
Failure Rate						
Tolerance Capability						
Type						
Breakdown						
Characteristics						

Class, Attribute Name	ASURE	Cost Advantage	Factor Aim	IGRIP	Quest	VSA
Resource/Tools/Breakdown						
Date Time						
Description						
Name						
Repair Time						
Time Between Failures						
Time to First Failure						
Resource/Tools/Breakdown/Resource See Resource Information						
Resource/Tools/Characteristics						
Date Time						
Description						
Name						
Numerical Value						
Text Value						

Notes:

- Object nesting is depicted with color-coding as well as with specification of the full object name, including all parent objects.
- Attributes are in the list below the interface name and are shown in alphabetical order.
- Methods are not identified in this matrix.
- I/O identifies that the information can be either input into the tool or output from the tool. In some cases I/O may refer to editing capability from the tool's interface, but not its use within the simulation tool itself.
- I identifies that the information can only be input into the tool.
- O identifies that the information is only a tool output.

APPENDIX D – Sample Code

This appendix contains sample C++ code for both the SDM client and the Simulator Work Flow Manager server interface software, the two elements that are necessary to make a tool SAVE-compliant.

SDM Sample Client

```
// Client.cpp
// A client for SAVE.

#include "save.hh"
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>


int main (int argc, char **argv)
{
  msmBaseObject_var msmBaseObjectVar;
  msmDbAccess_var msmDbAccessVar;
  msmLibrary_var msmLibraryVar;
  msmObjectSeq_var msmObjectSeqVar;
  msmOperation_var msmOperationVar;
  msmPersonnel_var msmPersonnelVar;
  msmProcessPlan_var msmProcessPlanVar;
  msmResource_var msmResourceVar;
  msmResourceApplic_var msmResourceApplicVar;
  msmResourcePool_var   msmResourcePoolVar;
  msmRiskInfo_var msmRiskInfoVar;
  msmSchedInfo_var msmSchedInfoVar;
  msmSimModel_var msmSimModelVar;
  msmSimReqst_var msmSimReqstVar;
  msmVersionedFloat_var msmVersionedFloatVar;
  msmVersionedString_var msmVersionedStringVar;

  if (argc < 2)
  {
     cout << "usage: " << argv[0] << " <hostname>" << endl;
     exit (-1);
  }

  // bind to the msmDbAccess object
  try
  {
      cout << "Binding to DbAccess object" << endl;
      msmDbAccessVar = msmDbAccess::_bind(":save", argv[1]);
  }
  catch (CORBA::SystemException &sysEx)
  {
    cerr << "Unexpected system exception" << endl;
    cerr << &sysEx;
    cout << "We are exiting when trying to bind to msmDbAccess object" <<
endl;
    exit(1);
  }
  catch(...)
  {
    // an error occurred while trying to bind to the msmDbAccess object.
    cerr << "Bind to object failed" << endl;
```

```
    cerr << "Unexpected exception " << endl;
    exit(1);
  }

  try
  {
    msmDbAccessVar->BeginUpdateTrans();

// Create a SimModel Object
      cout << "Creating a SimReqst" << endl;
      ObjectType simreqstType = SimReqst;
      char * simreqstName = "SimReqst Name1";
      char * simreqstDesc = "SimReqst Desc1";
      msmBaseObjectVar = msmDbAccessVar->CreateObject(simreqstType,
simreqstName, simreqstDesc);
      msmSimReqstVar = msmSimReqst::_narrow(msmBaseObjectVar);

      cout << "Checking a ProcessPlan" << endl;
      msmProcessPlanVar = msmSimReqstVar->ProcessPlan();

      cout << "Returned ProcessPlan, lets check it" << endl;

      if(CORBA::is_nil(msmProcessPlanVar))
      {
            cout << "Nil ProcessPlan... " << endl;
      }

// Create a ProcessPlan Object, add it to the SimReqst
      ObjectType processplanType = ProcessPlan;
      char * processplanName = "ProcessPlan Name1";
      char * processplanDesc = "ProcessPlan Desc1";
      msmBaseObjectVar = msmDbAccessVar->CreateObject(processplanType,
processplanName, processplanDesc);
      msmProcessPlanVar = msmProcessPlan::_narrow(msmBaseObjectVar);

      msmSimReqstVar->ProcessPlan(msmProcessPlanVar);

// Create 3 Operation Objects, add them to the Operations sequence
      ObjectType operationType = Operation;
      char * operationName = "Operation Name1";
      char * operationDesc = "Operation Desc1";
      msmBaseObjectVar = msmDbAccessVar->CreateObject(operationType,
operationName, operationDesc);
      msmOperationVar = msmOperation::_narrow(msmBaseObjectVar);

      msmObjectSeqVar = msmProcessPlanVar->Operations();
      msmObjectSeqVar->AddObject(msmOperationVar);

      operationName = "Operation Name2";
      operationDesc = "Operation Desc2";
      msmBaseObjectVar = msmDbAccessVar->CreateObject(operationType,
operationName, operationDesc);
      msmOperationVar = msmOperation::_narrow(msmBaseObjectVar);
      msmObjectSeqVar->AddObject(msmOperationVar);

      operationName = "Operation Name3";
      operationDesc = "Operation Desc3";
```

```
      msmBaseObjectVar = msmDbAccessVar->CreateObject(operationType,
operationName, operationDesc);
      msmOperationVar = msmOperation::_narrow(msmBaseObjectVar);
      msmObjectSeqVar->AddObject(msmOperationVar);

    msmDbAccessVar->Commit();
  }
  catch (msmDbAccess::msmInvalidObjectType &invalidObjectType)
  {
    cerr << "Object type not supported" << endl;
    exit(1);
  }
  catch (msmVersionedString::msmNoValidValue &dbLocked)
  {
    cerr << "No valid datetime in database" << endl;
      msmDbAccessVar->Commit();
    exit(1);
  }

  catch (CORBA::SystemException &sysEx)
  {
    cerr << "Unexpected system exception" << endl;
    cerr << &sysEx;
    exit(1);
  }
  catch(...)
  {
    cerr << "Unexpected exception " << endl;
    exit(1);
  }


  try
  {
    msmDbAccessVar->BeginReadTrans();

// Get the SimReqst by calling GetSimReqst
      char * simreqstName = "SimReqst Name1";
      msmSimReqstVar = msmDbAccessVar->GetSimReqst(simreqstName);
      cout << "SimReqst Name = " << msmSimReqstVar->Name() << endl;

// Get the SimReqst by calling GetLibrary, either way is fine
      char * libraryName = "SimReqst";
      msmLibraryVar = msmDbAccessVar->GetLibrary(libraryName);
      msmObjectSeqVar = msmLibraryVar->LibraryList();
      CORBA::Long firstObject = 1;
      msmBaseObjectVar = msmObjectSeqVar->GetByIndex(firstObject);
      msmSimReqstVar = msmSimReqst::_narrow(msmBaseObjectVar);
      cout << "SimReqst Name = " << msmSimReqstVar->Name() << endl;

// Check the ProcessPlan we associated with the SimReqst
      msmProcessPlanVar = msmSimReqstVar->ProcessPlan();
      if(CORBA::is_nil(msmProcessPlanVar))
      {
            cout << "Nil ProcessPlan " << endl;
      }
      else
```

```
      {
            cout << "ProcessPlan Name = " << msmProcessPlanVar->Name() <<
endl;
      }

// Three different ways of getting Operations from the ObjectSequence
      BaseObjectStruct myOperationsStruct;
      baseObjectStructSeq* myOperationsStructSeq;
      baseObjectSeq* myOperationsSeq;

      msmObjectSeqVar = msmProcessPlanVar->Operations();

// The objectStructSeq method, uses 1 Corba call
      myOperationsStructSeq = msmObjectSeqVar->objectStructSeq();
      CORBA::Long structSize = myOperationsStructSeq->length();
      cout << "The number of Operations = " << structSize << endl;
      for(int iLoop1 = 0; iLoop1 <= structSize-1; iLoop1++)
      {
            myOperationsStruct = (*myOperationsStructSeq)[iLoop1];
            cout << "The index = " << iLoop1+1 << " and the name = " <<
myOperationsStruct.structName << endl;
      }

// The objectSequence method, uses 1 Corba call to get the sequence of
BaseObjects
// and an additional CORBA call each time through the loop to get the Name (4
total)
      myOperationsSeq = msmObjectSeqVar->objectSequence();
      CORBA::Long seqSize = myOperationsSeq->length();
      cout << "The number of Operations = " << seqSize << endl;
      for(int iLoop2 = 0; iLoop2 <= seqSize-1; iLoop2++)
      {
            msmBaseObjectVar = (*myOperationsSeq)[iLoop2];
            msmOperationVar = msmOperation::_narrow(msmBaseObjectVar);
            cout << "The index = " << iLoop2 << " and the name = " <<
msmOperationVar->Name() << endl;
      }

// The Brute force method, uses 2 Corba calls each time through the loop,
// 1 for the GetByIndex and 1 for getting the Name (6 total)
      CORBA::Long sequenceSize = msmObjectSeqVar->NoInSeq();
      cout << "The number of Operations = " << sequenceSize << endl;

      for(CORBA::Long iLoop3 = 1; iLoop3 <= sequenceSize; iLoop3++)
      {
            msmBaseObjectVar = msmObjectSeqVar->GetByIndex(iLoop3);
            msmOperationVar = msmOperation::_narrow(msmBaseObjectVar);
            cout << "The index = " << iLoop3 << " and the name = " <<
msmOperationVar->Name() << endl;
      }

    msmDbAccessVar->EndReadTrans();
  }
  catch (msmDbAccess::msmInvalidObjectType &invalidObjectType)
  {
    cerr << "Object type not supported" << endl;
    exit(1);
```

```
  }
  catch (msmVersionedString::msmNoValidValue &dbLocked)
  {
    cerr << "No valid value/datetime in database" << endl;
      msmDbAccessVar->Commit();
    exit(1);
  }
  catch (CORBA::SystemException &sysEx)
  {
    cerr << "Unexpected system exception" << endl;
    cerr << &sysEx;
    exit(1);
  }
  catch(...)
  {
    cerr << "Unexpected exception " << endl;
    exit(1);
  }

  try
  {
    msmDbAccessVar->BeginUpdateTrans();

// Get the SimReqst by calling GetSimReqst
      char * simreqstName = "SimReqst Name1";
      msmSimReqstVar = msmDbAccessVar->GetSimReqst(simreqstName);
      msmProcessPlanVar = msmSimReqstVar->ProcessPlan();
      if(!CORBA::is_nil(msmProcessPlanVar))
      {
            // Note: Objects which are derived directly from msmBaseObject
            // are not created by the dbAccess->CreateObject() function and
            // are automatically created by the server.
            msmRiskInfoVar = msmProcessPlanVar->Risk();
            msmVersionedFloatVar = msmRiskInfoVar->FailureProbability();
            cout << "The CurrValue is " << msmVersionedFloatVar->CurrValue()
<< endl;
      }


      CORBA::Float currentValue = 55.5;
      char * currentSource = "CurrSource1";
      msmVersionedFloatVar->addNewValue (currentValue, currentSource);

      currentValue = 66.6;
      currentSource = "CurrSource2";
      msmVersionedFloatVar->addNewValue (currentValue, currentSource);

      currentValue = 77.7;
      currentSource = "CurrSource3";
      msmVersionedFloatVar->addNewValue (currentValue, currentSource);

      VersFloatStruct  myVersFloatStruct ;
      VersFloatStructSeq* myVersFloatStructSeq;
      myVersFloatStructSeq = msmVersionedFloatVar->VersionedFloatStructSeq();

      CORBA::Long structSize = myVersFloatStructSeq->length();
      cout << "The number of myVersFloats = " << structSize << endl;
```

```
      for(int iLoop = 0; iLoop <= structSize-1; iLoop++)
      {
            myVersFloatStruct = (*myVersFloatStructSeq)[iLoop];
            cout << "The index = " << iLoop+1 << endl;
            cout << "The Value = " << myVersFloatStruct.structCurrValue <<
endl;
            cout << "The Source = " << myVersFloatStruct.structCurrSource <<
endl;
            cout << "The DateTime = " << myVersFloatStruct.structCurrDateTime
<< endl;
      }

    msmDbAccessVar->Commit();
  }
  catch (msmDbAccess::msmInvalidObjectType &invalidObjectType)
  {
    cerr << "Object type not supported" << endl;
    exit(1);
  }
  catch (msmVersionedString::msmNoValidValue &dbLocked)
  {
    cerr << "No valid value/datetime in database" << endl;
      msmDbAccessVar->Commit();
    exit(1);
  }
  catch (CORBA::SystemException &sysEx)
  {
    cerr << "Unexpected system exception" << endl;
    cerr << &sysEx;
    exit(1);
  }
  catch(...)
  {
    cerr << "Unexpected exception " << endl;
    exit(1);
  }

  try
  {
    msmDbAccessVar->BeginUpdateTrans();

// Do some testing with ResourceApplic and Matrices...
// Create a ResourceApplic object
      ObjectType resourceapplicType = ResourceApplic;
      char * resourceapplicName = "ResourceApplicName";
      char * resourceapplicDesc = "ResourceApplicDesc";
      msmBaseObjectVar = msmDbAccessVar->CreateObject(resourceapplicType,
resourceapplicName, resourceapplicDesc);
      msmResourceApplicVar = msmResourceApplic::_narrow(msmBaseObjectVar);

      Matrix myMatrix;

      myMatrix[0][0] = myMatrix[0][1] = myMatrix[0][2] = myMatrix[0][3] =
0.0;
      myMatrix[1][0] = myMatrix[1][1] = myMatrix[1][2] = myMatrix[1][3] =
1.0;
```

```
      myMatrix[2][0] = myMatrix[2][1] = myMatrix[2][2] = myMatrix[2][3] =
2.0;
      myMatrix[3][0] = myMatrix[3][1] = myMatrix[3][2] = myMatrix[3][3] =
3.0;
      msmResourceApplicVar->TranformMatrix(myMatrix);

      msmObjectSeqVar = msmOperationVar->ToolResApplic();
      msmObjectSeqVar->AddObject(msmResourceApplicVar);

// Now read out the Matrix

      Matrix_slice      *mat_slice;
      CORBA::Float      ww, xx, yy, zz;

      mat_slice = msmResourceApplicVar->TranformMatrix();

      ww = mat_slice[0][0];
      xx = mat_slice[1][1];
      yy = mat_slice[2][2];
      zz = mat_slice[3][3];

      cout << "mat_slice[3][0] = " << ww << endl;
      cout << "mat_slice[3][0] = " << xx << endl;
      cout << "mat_slice[3][0] = " << yy << endl;
      cout << "mat_slice[3][0] = " << zz << endl;

    msmDbAccessVar->Commit();
  }
  catch (msmDbAccess::msmInvalidObjectType &invalidObjectType)
  {
    cerr << "Object type not supported" << endl;
    exit(1);
  }
  catch (msmVersionedString::msmNoValidValue &dbLocked)
  {
    cerr << "No valid value/datetime in database" << endl;
      msmDbAccessVar->Commit();
    exit(1);
  }
  catch (CORBA::SystemException &sysEx)
  {
    cerr << "Unexpected system exception" << endl;
    cerr << &sysEx;
    exit(1);
  }
  catch(...)
  {
    cerr << "Unexpected exception " << endl;
    exit(1);
  }


  try
  {
    msmDbAccessVar->BeginUpdateTrans();

// Get the ProcessPlan out of the library
```

```
      char * libraryName = "ProcessPlan";
      msmLibraryVar = msmDbAccessVar->GetLibrary(libraryName);
      msmObjectSeqVar = msmLibraryVar->LibraryList();
      char * processplanName = "ProcessPlan Name1";
      CORBA::Long strtindx = 1;
      CORBA::Long objindx;
      msmBaseObjectVar = msmObjectSeqVar-
>FindObject(processplanName,strtindx,objindx);
      msmProcessPlanVar = msmProcessPlan::_narrow(msmBaseObjectVar);

// Get the first msmOperation from the Operations sequence in this
ProcessPlan
      msmObjectSeqVar = msmProcessPlanVar->Operations();
      CORBA::Long firstObject = 1;
      msmBaseObjectVar = msmObjectSeqVar->GetByIndex(firstObject);
      msmOperationVar = msmOperation::_narrow(msmBaseObjectVar);

// Create a ResourceApplic to be used further down
      ObjectType resourceapplicType = ResourceApplic;
      char * resourceapplicName = "ResourceApplicName";
      char * resourceapplicDesc = "ResourceApplicDesc";
      msmBaseObjectVar = msmDbAccessVar->CreateObject(resourceapplicType,
resourceapplicName, resourceapplicDesc);
      msmResourceApplicVar = msmResourceApplic::_narrow(msmBaseObjectVar);

// A ProcessPlan contains many Personnel and Tool objects that are contained
in the
// msmPersonnelPool and msmToolPool ObjectSequences. A msmOperation has a
PersonResApplic
// and a ToolResApplic contained in it. Any Operation in a ProcessPlan can
draw from
// one or more ResourcePools associated with its ProcessPlan = each use is a
ResourceApplic

// Demonstrating use of a ProcessPlan<-->PersonnelPool<-->Personnel<--
>Resource relationship

// Create a msmPersonnel object, cast it to a msmResource object, put it in
the PersonnelPool
// ObjectSequence within the ProcessPlan

      ObjectType personnelType = Personnel;
      char * personnelName = "PersonnelName";
      char * personnelDesc = "PersonnelDesc";
      msmBaseObjectVar = msmDbAccessVar->CreateObject(personnelType,
personnelName, personnelDesc);
      msmPersonnelVar = msmPersonnel::_narrow(msmBaseObjectVar);

      ObjectType resourcepoolType = ResourcePool;
      char * resourcepoolName = "ResourcePoolName";
      char * resourcePoolDesc = "ResourcePoolDesc";
      msmBaseObjectVar = msmDbAccessVar->CreateObject(resourcepoolType,
resourcepoolName, resourcePoolDesc);
      msmResourcePoolVar = msmResourcePool::_narrow(msmBaseObjectVar);

// Note this msmResource is really a Personnel
      msmResourceVar = msmPersonnel::_duplicate(msmPersonnelVar);
```

```
// And this msmResourcePool is really a PersonnelPool
      msmResourcePoolVar->Resource(msmResourceVar);
// This should be placed in the PersonnelPool sequence in the ProcessPlan
      msmObjectSeqVar = msmProcessPlanVar->PersonnelPool();
      msmObjectSeqVar->AddObject(msmResourcePoolVar);

// And likewise this msmResourceApplic is for Personnel types only
      msmResourceApplicVar->Pool(msmResourcePoolVar);
// And therefore they get placed in the PersonResApplic and never in the
ToolResApplic
      msmOperationVar->PersonResApplic()->AddObject(msmResourceApplicVar);

    msmDbAccessVar->Commit();
  }
  catch (msmDbAccess::msmInvalidObjectType &invalidObjectType)
  {
    cerr << "Object type not supported" << endl;
    exit(1);
  }
  catch (msmVersionedString::msmNoValidValue &dbLocked)
  {
    cerr << "No valid value/datetime in database" << endl;
      msmDbAccessVar->Commit();
    exit(1);
  }
  catch (CORBA::SystemException &sysEx)
  {
    cerr << "Unexpected system exception" << endl;
    cerr << &sysEx;
    exit(1);
  }
  catch(...)
  {
    cerr << "Unexpected exception " << endl;
    exit(1);
  }

  return 0;
}
```

Simulator Sample Server

```
//Simulator_i.cpp

// --------------


#include <iostream.h>
#include <stdlib.h>

#include "Simulator_i.h"

/**
 * Simulator implementation constructor. It saves the name for use in
 * the listener change state callback. The constructor creates a single
 * property called "SimRequest", which is required by the Work Flow Manager.
 * It sets the initial state to UNDEFINED and zeros out the listeners.
 */
Simulator_i::Simulator_i(char* name)
{
    // Set the simulator name. This is the name used to register the
    // simulator with the ORB. It serves as the name parameter in
    // the listener calls.
    simulatorName = new char[strlen(name) + 1];
    strcpy(simulatorName, name);

    // Set the default simulator state. All simulators start in the
    // undefined state.
    simState = UNDEFINED;

    // Set the simulator properties. The only property required is the
    // "SimRequest" property, which the Work Flow Manager loads with the
    // name of the SimReqst object.
    char* nameStr = CORBA::string_alloc(strlen("SimRequest") + 1);
    char* valueStr = CORBA::string_alloc(strlen("simrequest") + 1);
    char* describStr = CORBA::string_alloc(strlen("A simulation request.") +
1);
    strcpy(nameStr, "SimRequest");
    strcpy(valueStr, "simrequest");
    strcpy(describStr, "A simulation request.");

    // Create a property from the strings.
    Property* property = new Property();
    property->name = nameStr;
    property->value = valueStr;
    property->description = describStr;
    property->type = STRING_TYPE;

    // Create the property sequence and add the property to it.
    numProperties = 1;
    propertySeq = new PropertySeq(numProperties);
    propertySeq->length(numProperties);
    (*propertySeq)[0] = *property;
```

```
    // Set the current number of listeners. The listener array holds the
names
    // of the registered listeners and is dynamically allocated in the add
    // and remove methods.
    numListeners = 0;
    listeners = new char*[0];
}

/**
 * The destructor destroys the listener array. This reclaims the memory
 * on the way out.
 */
Simulator_i::~Simulator_i() {
    delete [] listeners;
}

/**
 * Acknowledge send from a client that it received a user changed state from
 * the simulator. What the simulator does with it is up to it. This one
 * does nothing with it since it is not used by the Work Flow Manager.
 *
 * @param state  the state to acknowledge
 * @param IT_env the CORBA environment
 */
void Simulator_i::ackUserChangedState(SimulatorState state,
                                      CORBA::Environment &IT_env)
    throw (CORBA::SystemException)
{
      cout << "Entering ackUserChangedState" << endl;
}

/**
 * Sets the properties for the simulator. It takes each property and
 * sets it. It throws exceptions if the property is invalid or does
 * not exist. The only property in this server is "SimRequest", so
 * any other ones will cause an exception. Note that a partial list
 * of properties are accepted by this method and it updates only those
 * properties in the list.
 *
 * @param properties the sequence of properties
 * @param IT_env     the CORBA environment
 */
void Simulator_i::setProperties(const PropertySeq& properties,
                        CORBA::Environment &IT_env)
    throw (CORBA::SystemException, Simulator::NoSuchProperty,
         Simulator::InvalidProperty)
{
      cout << "Entering setProperties" << endl;
    char*           name;
    PropertyType    type;
    int             found;
    Property*       property;
    const Property* foundProperty;

    // Create a temporary property sequence from the current properties.
    // This way, it can be thrown away if there is an invalid property
    // in the sequence. Since the incoming sequence does not have to
```

```
    // contain all the properties, you cannot just copy it into the
    // existing sequence or you may loose properties.
    PropertySeq tempPropertySeq = *propertySeq;

    // Check the properties to verify that the input contains valid
properties.
    // A valid property must match both name and type.
    for (int i = 0; i < properties.length(); i++)
      {
            foundProperty = &properties[i];
            name = CORBA::string_dup(foundProperty->name);
            type = foundProperty->type;
            found = 0;

            // Scan the temporary sequence and look for a match. By breaking
when
            // a match is found, the property variable holds the temporary
            // property to modify.
            for (int j = 0; j < numProperties; j++)
            {
                  property = &tempPropertySeq[j];
                  if (!strcmp(name, CORBA::string_dup(property->name)) &&
                              (type == property->type))
                  {
                        found = 1;
                        break;
                  }
            }

            // If found, set the temporary property value to match the found
one.
            // Otherwise, throw an exception.
            if (found)
                  property->value = foundProperty->value;
            else
                  throw Simulator::NoSuchProperty(name);
    }

    // All the new properties were valid so the old sequence can now be
    // replaced with the temporary one. The sequence is deep copied with
    // the overloaded equals.
    *propertySeq = tempPropertySeq;
}

/**
 * Returns the sequence of properties. These are deep copied into a
 * new object so the client may delete it without affecting the
 * internal copy.
 *
 * @param IT_env the CORBA environment
 *
 * @return The sequence of properties.
 */
PropertySeq* Simulator_i::getProperties(CORBA::Environment &IT_env)
    throw (CORBA::SystemException)
{
      cout << "Entering getProperties" << endl;
```

```
    PropertySeq* propSeq = new PropertySeq(*propertySeq);
    return propSeq;
}

/**
 * Get the platform object. This version creates a platform structure and
 * fills it with some arbitrary strings. The Work Flow Manager does not
 * currently use the platform.
 *
 * @param IT_env the CORBA environment
 *
 * @return The platform information.
 */
Platform* Simulator_i::getPlatform(CORBA::Environment &IT_env)
    throw (CORBA::SystemException)
{
    char* nameStr = CORBA::string_alloc(strlen("name") + 1);
    char* ipStr = CORBA::string_alloc(strlen("addr") + 1);
    char* osStr = CORBA::string_alloc(strlen("os") + 1);
    strcpy(nameStr, "name");
    strcpy(ipStr, "addr");
    strcpy(osStr, "os");
    Platform* platform = new Platform();
    platform->hostName = nameStr;
    platform->hostIPAddr = ipStr;
    platform->operSys = osStr;
    return platform;
}

/**
 * Returns the documentation URL string for the simulator. This is an
 * empty string at the moment. The Work Flow Manager does not currently
 * use the URL. Nulls cannot be returned so the string is empty, not null.
 *
 * @param IT_env the CORBA environment
 *
 * @return The documentation string.
 */
char* Simulator_i::getDocumentation(CORBA::Environment &IT_env)
    throw (CORBA::SystemException)
{
    char* simURL = CORBA::string_alloc(1);
    strcpy(simURL, "");
    return simURL;
}

/**
 * Returns the operations supported by this simulator. This particular
 * simulator contains two operations: LAUNCH and TERMINATE.
 *
 * @param IT_env     the CORBA environment
 *
 * @return The sequence of operations.
 */
SimulatorOperationSeq* Simulator_i::getOperations(CORBA::Environment &IT_env)
    throw (CORBA::SystemException)
{
```

```
      cout << "Entering getOperations" << endl;
    SimulatorOperationSeq* opnArray = new SimulatorOperationSeq(2);
    opnArray->length(2);
    (*opnArray)[0] = LAUNCH;
    (*opnArray)[1] = TERMINATE;
    return opnArray;
}

/**
 * Returns the commands for the specified operation. It throws an
 * exception if the operation does not exist. The LAUNCH operation
 * has the commands "import" and "export" while the TERMINATE operation
 * has no commands.
 *
 * @param operation the operation type
 * @param IT_env    the CORBA environment
 *
 * @return The sequence of command strings.
 */
StringSeq* Simulator_i::getCommands(SimulatorOperation operation,
                                    CORBA::Environment &IT_env)
    throw (CORBA::SystemException, Simulator::NoSuchOperation)
{
      cout << "Entering getCommands" << endl;
    StringSeq* strs = new StringSeq();
    char*      importStr;
    char*      exportStr;

    // There are two supported operations for this simulator; LAUNCH and
    // TERMINATE. The LAUNCH has two commands. The TERMINATE has no commands
    // and so returns an empty sequence. Any other operations throw an
    // exception.
    switch(operation) {
    case LAUNCH:
      importStr = CORBA::string_alloc(strlen("import" + 1));
      strcpy(importStr, "import");
      exportStr = CORBA::string_alloc(strlen("export" + 1));
      strcpy(exportStr, "export");
      strs->length(2);
      (*strs)[0] = importStr;
      (*strs)[1] = exportStr;
      break;

    case TERMINATE:
      strs->length(0);
      break;

    default:
      throw Simulator::NoSuchOperation();
    }
    return strs;
}

/**
 * Performs the specified operation with the specified command. If the
 * operation or command is not valid, it throws an exception. In this
 * implementation, a LAUNCH operation changes the state from UNDEFINED to
```

```
 * ENABLED to WORKING. A TERMINATE operation changes the state to
 * TERMINATED. Since this method is defined as oneway, the server can
 * block while it performs the operation, but it must still invoke the
 * state change callbacks to the registered listeners as it works. If it
 * gets an invalid operation or command, the server should immediately
 * change it state to FAULTY to inform the listeners of the failure.
 *
 * @param operation the operation type
 * @param command   the command string for the operation
 * @param IT_env    the CORBA environment
 */
void Simulator_i::doOperation(SimulatorOperation operation,
                        const char * command,
                        CORBA::Environment &IT_env)
    throw (CORBA::SystemException)
{
      cout << "Entering doOperation" << endl;
    // Immediately reject if the operation or command is null. It does this
    // by changing the state to FAULTY.
    if ((operation == NULL) || (command == NULL)) {
      changeState(FAULTY, IT_env);
      return;
    }

    // Check the operations by their type and/or command. If it is invalid,
    // change the state to FAULTY. If it is valid, perform the operation.
    switch (operation) {
    case PREPARE:
    case PAUSE:
    case RESUME:
      changeState(FAULTY, IT_env);
      break;

    case LAUNCH:
      if (strcmp(command, "import") && strcmp(command, "import"))
          changeState(FAULTY, IT_env);
      else {
          changeState(ENABLED, IT_env);
          changeState(WORKING, IT_env);
      }
      break;

    case TERMINATE:
      if (strcmp(command, ""))
          changeState(FAULTY, IT_env);
      else
          changeState(TERMINATED, IT_env);
      break;

    default:
      break;
    }
}

/**
 * Returns the current state of the simulator. This is the internal state
 * at any given time. It must return immediately.
```

```
 *
 * @param IT_env the CORBA environment
 *
 * @return The current simulator state.
 */
SimulatorState Simulator_i::getState(CORBA::Environment &IT_env)
    throw (CORBA::SystemException)
{
      cout << "Entering getState" << endl;
     return simState;
}

/**
 * Create an array for the listeners. Any current listeners are copied to
 * the array and the new listener is added to the end. The old array is
 * deleted to recover the memory. If the listener is already in the
 * array, nothing happens. The names of the listeners must be used
 * since the object pointers change within Orbix.
 *
 * @param listener the listener object pointer
 * @param IT_env   the CORBA environment
 */
void Simulator_i::addSimulatorListener(SimulatorListener_ptr listener,
                                       CORBA::Environment &IT_env)
    throw (CORBA::SystemException, Simulator::ListenersAtMaximum)
{
      cout << "Entering addSimulatorListener, listener = " << listener <<
endl;
    // Scan the current array and if the listener is already in it, leave.
    char* name = listener->_object_to_string();
    for(int i = 0; i < numListeners; i++)
      if (!strcmp(listeners[i], name))
          return;

    // Create a new listener array and copy the current listeners into it.
    char** newList = new char*[numListeners + 1];
    for(i = 0; i < numListeners; i++)
      newList[i] = listeners[i];

    // Add the new listener to the end of the new list.
    char* newListener = new char[strlen(name) + 1];
    strcpy(newListener, name);
    newList[numListeners] = newListener;

    // Delete the old array, assign the new one to it and increment the
    // number of listeners.
    delete [] listeners;
    listeners = newList;
    numListeners++;
}

/**
 * Remove a registered listener from the simulator listeners list. The array
 * of listener names is copied into a new array minus the listener to remove.
 * The old array is then deleted to recover the memory. If the listener is
not
 * in the array, nothing happens. The names of the listeners must be used
```

```
 * since the object pointers change within Orbix.
 *
 * @param listener the listener to remove
 * @param IT_env   the CORBA environment
 */
void Simulator_i::removeSimulatorListener(SimulatorListener_ptr listener,
                                  CORBA::Environment &IT_env)
    throw (CORBA::SystemException)
{
      cout << "Entering removeSimulatorListener" << endl;
    // Scan the current array and if the listener is not in it, leave.
    int found = 0;
    char* name = listener->_object_to_string();
    for(int i = 0; i < numListeners; i++) {
      if (!strcmp(listeners[i], name)) {
          found = 1;
          break;
      }
    }
    if (!found)
      return;

    // Create a new listener array and, with the exception of the one to
    // remove, copy the current listeners into it. Delete the one to remove.
    char** newList = new char*[numListeners - 1];
    int index = 0;
    for(i = 0; i < numListeners; i++)
      if (strcmp(listeners[i], name))
          newList[index++] = listeners[i];
      else
          delete [] listeners[i];

    // Delete the old array, assign the new one to it and decrement the
    // number of listeners.
    numListeners--;
    delete [] listeners;
    listeners = newList;
}

/**
 * Changes the internal state and notifies all registered listeners of
 * the change. This is not a part of the IDL description but is here to
 * handle the listeners. It does this by taking each listener name,
 * converting it to an object, narrowing the object to a SimulatorListener
 * and then invoking the simulatorChangedState() method in it. If an
 * exception is thrown, it stops updating the listeners.
 *
 * @param state  the new state of the simulator
 * @param IT_env the CORBA environment
 */
void Simulator_i::changeState(SimulatorState state, CORBA::Environment
&IT_env)
{
      cout << "Entering changeState" << endl;
    // Set the new state.
    simState = state;
```

```
    // Get the reference for each listener and invoke the change state
method.
      CORBA::Object* object;
    SimulatorListener_ptr listener;
    try
      {
            for(int i = 0; i < numListeners; i++)
            {
                  object = CORBA::Orbix.string_to_object(
                              CORBA::string_dup(listeners[i]),IT_env);
                  if (object != NULL)
                  {
                        listener = SimulatorListener::_narrow(object);
                        if (listener != NULL)
                              listener->simulatorChangedState(simState,
simulatorName,
                                          IT_env);
                  }
            }
    }
      catch (CORBA::SystemException &IT_exSys) { }
}
```

APPENDIX E

CORBA IDL for SAVE Data Model and Work Flow Manager

E-1 SAVE Data Model

Note: This MS Word file is fomatted to allow it to be written to a simple text file and compliled.

```
/*
Simulation Assessment Validation Environment (SAVE)
SAVE Manufacturing Simulation Model ( MSM )
CORBA IDL Final Demonstration Release
Version 3.0.3
April 23, 1999

Distribution Statement A:
Approved for public release; Distribution is unlimited.
*/

/**
This model contains the interface definitions for the SAVE data model used to
integrate manufacturing simulation tools.  The SAVE system is designed to
support manufacturing design studies, generating cost, schedule, and risk
estimates for product and process design alternatives.
*/

/*
Summary of Changes

Version 3.0.3

Added status flag to msmProcessPlan. When a process plan status is either
review or released, all objects associated with the process plan are locked
with the exception of reference type data that is typically stored in a
library.  These include Break, Breakdown, CAD Model, Inflation Table,
Material, Manufacturing Program, Part, Personnel, Reference Process, Tool,
Work Calendar and Work Shift.

Added msmStatusLock exception to list in msmDbAccess.  This exception is
raised whenever the status of an object is either review or released.  In
these cases, the object may not be modified.  If modification is necessary,
the status of the object that controls the object status must be set to
working.  Currently only msmDesignStudy, msmDesignAlternative, and
msmProcessPlan have status flags that may be set by the user.

Added a CopyObject method to msmDbAccess.  This gives the client the
capability to copy certain objects.  Currently, any library object may be
copied.  The client must supply a name and description for the new object.
Attributes and lower level objects that are associated with a copied object
will also be copied or used as appropriate.  Generally, reference data will
be used instead of copied.
```

Version 3.0.2

Added three attributes to msmCostInfo to account for the inflation data for specific cost information. These include BaseYear, MaterialInflFactor, and LaborInflFactor. Modified the definition of FiscalYear to reflect the year for which inflation values are valid.

Modified definition for the precedence attribute in msmOperation.

Changes all instances of msmCADModel to msmObjectSeq. This change effected msmResource, msmPart, and msmSimModel. This change provides a sequence of CAD models associated with these items so that different model formats for the same entity may be used. For example, one tool may need the CAD in native format, while another needs it converted into IGES format.

Modified definition of type attribute in msmTool to distinguish between stationary and moveable tools.

Added an attribute to msmObjectSeq that identifies the type of objects that are in the sequence. This attribute is readonly and is automatically set when the object sequence is initially created.

The AddObject and InsertBefore methods in msmObjectSeq will perform checking for duplicate names in object sequences specified in the server configuration file to require unique names. If the object has the same name as another object in the sequence, the object is not added to the sequence and the msmDuplicateObject exception is raised. Added a readonly attribute that allows checking to see if duplicates are allowed for a particular object sequence.

The constructor for msmWorkCalendar will automatically create a calendar for the current year, the WorkYear attribute is used to change the calendar year. The Calendars attribute in msmPersonnel is now an msmObjectSeq of msmWorkCalendars.

Modified DeleteObject method in msmObjectSeq to be RemoveObject. This method will remove an object from the specified sequence. It will not delete the object from the database.

Deleting objects will be handled by the server to insure that no objects are deleted while they are still in use somewhere. Each object will contain an internal counter that increments or decrements each time it is used or removed from use. When an object's internal counter goes to zero, it will delete itself.

Version 3.0.1

Modified part attribute in msmProcessPlan to include a msmPart for the Engineering Bill of Materials (EBOM) and another for the Manufacturing Bill of Materials (MBOM).

Changed msmRiskInfo to a msmNamedObject in order to allow specification of the type of risk stored in the object. The name attribute inherited from msmNamedObject should be used for this purpose. Modified use of Risk in msmProcessPlan, msmOperation, and msmRefProcess to a sequence of msmRiskInfo objects to allow different types of risk to be associated with each object.

The name attribute specified the type of risk; therefore, duplicate names are allowed in the sequences.

Changed PercentContribution attribute in msmContributor object to a type of msmVersionedFloat to allow versioning of contributor data along with the associated risk data.

The msmProcessPlan object has been modified to contain a sequence of msmMfgOrder objects to account for multiple manufacturing order associated with a single process plan.

A msmSchedInfo object has been added to the msmMfgOrder object in order to provide planned schedule information for a specific manufacturing order.

Added two interfaces associated with parts: msmPartUsage and msmPartLocation. These interfaces will specify the quantity of a part in a given operation and their transformation matrices, respectively. The quantity attribute was removed from msmPart, and the msmPart interface was added to the library. Modified part sequence in msmOperation to include one msmPartUsage sequence for consumed parts and another for produced parts. Forced consistency between name attribute in msmPartUsage and the number attribute in its msmPart. Forced consistency between the quantity in msmPartUsage and the number of msmPartLocations in the sequence.

Added msmDesignAlternative to the library.

Version 2.0

The CreateObject method has been moved from msmObjectFactory to msmDbAccess. The msmObjectFactory object has been deleted.

A sequence of base objects was added to allow retrieval of all objects in a msmObjectSeq at one time.

A structure was added (BaseObjectStruct) that contains the name and description of any msmBaseObject. This structure is used to return a sequence of all objects in a msmObjectSeq along with their names and descriptions at one time.

Structures were added (VersStringStruct and VersFloatStruct) that contain the value, source, and datetime for a particular versioned variable. This structure is used to return a sequence of the versioned values for a particular attribute.

Simulation Requests have been added to the Library object.

The msmBreak object has been changed to a named object and added to the library.

Constructors will automatically create base objects that are attributes of other objects. These should be used by client codes for population of data in lieu of creating new ones. The capability to create base objects (msmCostInfo, msmRiskInfo, msmScheduleInfo, msmValueWithUnits, msmVersionedFloat, and msmVersionedString) using the CreateObject method has been removed to prevent overwriting of the existing objects. As an additional safety, these objects are made "readonly attributes" of their parent object.

Constructors will not automatically create named objects that are attributes of other objects. It is the responsibility of the clients to check using the CORBA: IS_NIL() function for an existing object reference to determine whether to create the object or use the existing one. The rationale for this approach is that many named objects are libraries and may be used numerous times within the model. Automatic creation by the constructors could instantiate objects that would never be used.

If any conflicts exist between these notes and the ones below for previous versions, the Version 2.0 note applies.

Previous Versions

A containing object (msmObjectSeq) that provides operations to perform necessary function on the sequence has replaced all sequences.

Objects have replaced all structures in order to allow for easier inclusion in a containing (sequence) object.

A factory object has been added to explicitly control creation of any new objects.

Wherever possible, objects are created automatically by the constructor of theobject in which they are attributes. For example, the msmCostinfo, msmSchedinfo, and msmRiskInfo objects are created when the msmOperation object in which they reside is created. These automatic object creations are noted in comments in the creating object.

Library objects automatically add themselves to the appropriate msmLibrary object. This fact is noted in comments in these objects.

The Runtime attribute in msmOperation is defined as being in units of Hours.

ProcessPlans have been added to a Library.

Attributes of the Queue structure are now directly attributes of msmOperation.

Representation of an Inflation Table is changed.

The msmCharacteristic object is now a single name/value pair and is sequenced where needed by msmObjectSeqAdded date/time operations for increments.

Added comments to define which attribute objects are created by the constructor. All other attribute objects must be created by the Factory object or found for use in a Library.

Notes on Object Constructors

Constructors will automatically create and populate the date-time attribute for all named objects.

Constructors will automatically create empty object sequences where they exist within an interface.

Constructors will automatically create "simple" attributes of objects. Where noted in the comments, these attributes are initialized.

Configuration Management of SAVE Object Model

Configuration management of data within this model is provided at several levels.

Design Studies and their Alternatives have a Status attribute that can be set as Working, Review, or Released Process Plans are versioned (as they are defined in the Design Study Alternative)

Many variables are defined as Versioned Variables (string or numeric) allowing new estimates for a value to be added without losing the previous value. The latest value is used by default, but previous versions can be obtained by date/time stamp

Initialization of Interfaces

Default constructors are not shown in this version of the IDL. The IDL compiler automatically generates these. In cases where initialization of variables is necessary in the creation of an interface, a specific constructor is written for that purpose.

Naming Conventions in This Model

All interface names are prefaced with "msm" to indicate that they are part of the Manufacturing Simulation Model. ex: msmProcessPlan

Exceptions in This Model

Some exceptions are defined, but are not raised in declared operations. These exceptions can be raised by the methods that implement the attribute read/write access.

Model Flexibility With Use of Characteristics

A msmCharacteristic interface is defined and included in some objects to allow a list of Name / Value pairs to be defined. This allows great flexibility in user-defined attributes beyond the standard attributes defined in the model. Lists of msmCharacteristic are provided by msmObjectSeq.

Libraries of Reference Information in Manufacturing Simulation Model

A library object is defined and will be instantiated for reference information that can used by clients and users to assure consistency in creating other objects. A library is simply a list of MSM objects of a particular type, and provides easy access to all objects of that type. Library objects will be created during the initial database creation process.

Typically these libraries will be populated manually using the Query Manager, but other automated loading tools are possible, when written as MSM clients.

Note to server developers - Constructors for Library objects should add new objects to the library. The following libraries are defined:

```
Break
CAD Models
Design Alternatives
Design Studies
Inflation Tables
Materials
MfgProgram
Part
Personnel
ProcessPlan
Reference Processes
Simulation Request
Tools
WorkCalendar
WorkShift
*/
```

```
//-------------------------------------------------------------------------
-
//    Enumerations
//-------------------------------------------------------------------------
-

/**
Status (Working / Review / Released).
*/
enum Status { Working, Review, Released } ;

/**
Type of part.
*/
enum PartType { Detail, Assmbly } ;

/**
Days of the week.
*/
enum  DaysOfWeek { Sun,Mon,Tue,Wed,Thu,Fri,Sat } ;

/**
Simulation code type.
*/
enum  SimType { Enterprise, CAD, ProcessPlanning, Schedule, Assembly,
Factory,
Cost, Risk, Variability } ;

/**
Cost type.
*/
enum CostType { FirstUnit, Average, Total } ;

/**
Qualitative rating - textual.
*/
enum  TextQual { low, med, high } ;

/**
Object types used in the manufacturing simulation model.
*/
enum ObjectType { BaseObject, Break, Breakdown, CADModel, Characteristic,
Contributor, CostInfo, DbAccess, DateTime, DesignAlternative, DesignStudy,
Feature, InflationTable, Library, Material, MfgOrder, MfgProgram,
NamedObject, ObjectFactory, ObjectSeq, Operation, Part, PartLocation,
PartUsage, Personnel,ProcessPlan, RefProcess, Resource, ResourcePool,
ResourceApplic, RiskInfo, SchedInfo, SimModel, SimReqst, Tool,
ValueWithUnits, VersionedFloat, VersionedString, WorkCalendar, WorkShift } ;
```

```
//------------------------------------------------------------------------
-
//    Interfaces
//------------------------------------------------------------------------
-

/*
Declare all Interfaces.
*/

interface   msmBaseObject;
interface   msmBreak;
interface   msmBreakdown;
interface   msmCADModel;
interface   msmCharacteristic;
interface   msmContributor;
interface   msmCostInfo;
interface   msmDbAccess;
interface   msmDateTime;
interface   msmDesignAlternative;
interface   msmDesignStudy;
interface   msmFeature;
interface   msmInflationTable;
interface   msmLibrary;
interface   msmMaterial;
interface   msmMfgOrder;
interface   msmMfgProgram;
interface   msmNamedObject;
interface   msmObjectSeq;
interface   msmOperation;
interface   msmPart;
interface   msmPartLocation;
interface   msmPartUsage;
interface   msmPersonnel;
interface   msmProcessPlan;
interface   msmRefProcess;
interface   msmResource;
interface   msmResourceApplic;
interface   msmResourcePool;
interface   msmRiskInfo;
interface   msmSchedInfo;
interface   msmSimModel;
interface   msmSimReqst;
interface   msmTool;
interface   msmValueWithUnits;
interface   msmVersionedFloat;
interface   msmVersionedString;
interface   msmWorkCalendar;
interface   msmWorkShift;
```

```
//---------------------------------------------------------------------------
-
//    Structures
//---------------------------------------------------------------------------
-

/**
Base Object structure that includes the name and description of any object
that inherits from msmBaseObject.
*/

struct BaseObjectStruct {

msmBaseObject structBaseObject ;
string structName ;
string structDesc ;
};

/**
Versioned string sturcture that contains the value, source, date, and time
for any attribute of type msmVersionedString.
*/

struct VersStringStruct {

string structCurrValue ;
string structCurrSource ;
string structCurrDateTime ;
};

/**
Versioned float structure that contains the value, source, date, and time for
any attribute of type msmVersionedFloat.
*/

struct VersFloatStruct {

float structCurrValue ;
string structCurrSource ;
string structCurrDateTime ;
};
```

```
//--------------------------------------------------------------------------
-
//    Type Definitions
//--------------------------------------------------------------------------
-

/**
Definitions for special types.
*/

typedef float Matrix [4] [4];

typedef sequence <msmBaseObject> baseObjectSeq ;

typedef sequence <BaseObjectStruct> baseObjectStructSeq ;

typedef sequence <VersStringStruct> VersStringStructSeq ;

typedef sequence <VersFloatStruct> VersFloatStructSeq ;

typedef long NumQual  ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    MANUFACTURING SIMULATION MODEL BASE OBJECT
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This is a base object of all SAVE objects in this model.  It defines the
methods common to all objects.
*/
interface  msmBaseObject  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

readonly attribute ObjectType objTyp;

attribute string SetRef;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    MANUFACTURING SIMULATION MODEL NAMED OBJECT
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This is a base object that includes Name, Description, and DateTime
attributes.
*/
interface  msmNamedObject : msmBaseObject {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Date and time of object creation.
*/
attribute  msmDateTime  DateTime ;

/**
Name of object.
*/
attribute string  Name ;

/**
Description of object.
*/
attribute string  Description ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    RESOURCE
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
This interface represents a resource utilized by a manufacturing process
operation.  This is a base class and will have Personnel and Tool interfaces
derived from it. Constructor will initialize efficiency to 0.
*/
interface  msmResource  :  msmNamedObject  {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

/**
Resource productivity  - % of available usage.
*/
attribute float Efficiency ;

/**
CAD models of resource.
*/
attribute msmObjectSeq CADModels;

//------------------------------------------------------------------------
-
//    Methods
//------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    BREAK
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface defines the start and end times associated with breaks in the
msmWorkShift. Constructor will automatically place this object in the
appropriate library.

*/
interface  msmBreak : msmNamedObject  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Break start time.
*/
attribute float  StartTime ;

/**
Break end time.
*/
attribute float  EndTime ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    BREAKDOWN
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface defines the breakdown (in terms of failure to operate)
characteristics of a tool.  It also contains the resource required to repair
the breakdown. The name and description attributes contain the mode, or
manner of failure of the breakdown.  The constructor will create the
RepairResource sequence and leave it empty.
*/
interface  msmBreakdown : msmNamedObject  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Mean time to first failure - Hours.
*/
attribute float  HrToFirstFail ;

/**
Indication of the amount of time between breakdowns for a given tool (mean
time between failure) in hours.
*/
attribute float  MHrBF ;

/**
Time to repair in hours.
*/
attribute float  RepairHr ;

/**
Resources required for repair.
*/
attribute msmObjectSeq  RepairResource ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    CAD MODEL
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface contains descriptive and location information about the
graphical representation of models used for simulations (includes parts,
tools, personnel, etc.). Constructor will automatically place this object in
the appropriate library.
*/
interface  msmCADModel : msmNamedObject {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
X dimension (height) of the envelope that would fully enclose the model -
Inches.
*/
attribute float EnvelopX ;

/**
Y dimension (width) of the envelope that would fully enclose the model -
Inches.
*/
attribute float EnvelopY ;

/**
Z dimension (depth) of the envelope that would fully enclose the model -
Inches.
*/
attribute float EnvelopZ ;

/**
Storage format for the CAD model. Could be native CAD or standards-based.
*/
attribute string Format ;

/**
Storage location for the CAD model. Computer node and path information.
*/
attribute string  Location ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-
```

```
//    None defined.
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    CHARACTERISTIC
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface contains a name / value / units that can be used to generally
expand the attributes in any interface.  Constructor creates numeric value.
*/
interface   msmCharacteristic  : msmNamedObject {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Textual values of characteristic.
*/
attribute string TextValue ;

/**
Numerical value with units.
*/
readonly attribute msmValueWithUnits  NumericValue ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//---------------------------------------------------------------------------
-
//---------------------------------------------------------------------------
-
//    CONTRIBUTOR
//---------------------------------------------------------------------------
-
//---------------------------------------------------------------------------
-

/**
This interface contains a contributor and its percentage to a RiskInfo
object. Constructor will intialize PercentContribution to 0.
*/
interface  msmContributor : msmNamedObject  {

//---------------------------------------------------------------------------
-
//    Attributes
//---------------------------------------------------------------------------
-

/**
Percentage of the total due to this contributor.
*/
readonly attribute msmVersionedFloat  PercentContribution ;

//---------------------------------------------------------------------------
-
//    Methods
//---------------------------------------------------------------------------
-

//   None defined.
}  ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    COST INFORMATION
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
This interface contains the basic set of cost data used throughout the model.
All values are in US dollars consistent with the "FiscalYear" value. All
variables are stored as msmVersionedFloat  types to allow versioning of the
data. The cost interface is used at various levels of the model.  Constructor
will create all msmVersionedFloat attributes.
*/
interface msmCostInfo : msmBaseObject  {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

/**
Type of cost being stored (FirstUnit, Average, Total). Will be defaulted to
Average by constructor.
*/
attribute CostType  Type;

/**
Year for which cost and inflation values are valid.
*/
attribute long  FiscalYear;

/**
Year for which the inflation value is 1.0.  This year must be the current
year or earlier.
*/
attribute long BaseYear;

/**
Factor used to inflate the material cost from the base year to the fiscal
year.
*/
attribute float MaterialInflFactor;

/**
Factor used to inflate the labor cost from the base year to the fiscal year.
*/
attribute float LaborInflFactor;

/**
Inflation table to allow for conversion among years.
*/
attribute msmInflationTable InflationTable;
```

```
/**
The repetitive manufacturing labor cost expended in the fabrication,
assembly, and test of a hardware product.  It includes field operations, mod
& test support.
*/
readonly attribute msmVersionedFloat RecurringMfgLabor;

/**
Recurring cost associated with such elements as travel, photographic
services, facility rental, multi-media, etc.
*/
readonly attribute msmVersionedFloat OtherRecurringMfg;

/**
Non-recurring cost associated with elements such as travel, in country
program office, overtime premium, reproduction, etc.
*/
readonly attribute msmVersionedFloat OtherNonrecurringMfg;

/**
Cost for raw materials, purchased parts, and standard hardware.
*/
readonly attribute msmVersionedFloat MaterialCost;

/**
Cost associated with activities such as process inspection, nondestructive
inspection, records keeping/maintenance, quality action reporting.
*/
readonly attribute msmVersionedFloat QualityAssurance;

/**
Cost associated with the tooling required during a product's development
phase.  The tooling concepts used generally are incomplete and/or low
production rate quality tools.
*/
readonly attribute msmVersionedFloat DevelopmentTooling;

/**
The cost of tooling that is generally made of the best and most practical
materials available. They are capable of producing parts with critical
tolerances at an accelerated production rate with relatively little to no
additions or changes in design.
*/
readonly attribute msmVersionedFloat ProductionTooling;

/**
Cost of fixtures, molds, jigs, electronic media, and mfg. Aids acquired or
manufactured by a contractor.
*/
readonly attribute msmVersionedFloat NonrecurringTooling;

/**
Material cost associated with the raw material required to construct the tool
inhouse or the supplier's material charge to construct the tool.
*/
readonly attribute msmVersionedFloat ToolMaterial;
```

```
/**
Cost associated with engineering and manufacturing tool maintenance. Some of
the activities within the engineering area are tool design, methods planning,
and liaison. Some of the activities within the manufacturing area are repair,
refurbish and incorporation of changes to existing tools, and tool storage.
*/
readonly attribute msmVersionedFloat SustainingTooling;

/**
Cost associated with capital equipment such as autoclaves, portable tools,
vehicles, and machinery that are used in the manufacture, research and
development, and/or test of products or general plant purpose.
*/
readonly attribute msmVersionedFloat PlantEquipment ;

/**
The cost of the mfg labor and materials required to build the product end
item.  This cost is generally a basic functional cost catagory that does not
include tooling cost.
*/
readonly attribute msmVersionedFloat TotalMfgCost;

/**
This cost is the arithmetic mean of the number of units to be built and
represents a cost of a unit that has not incurred any start up problems
typically associated with a production run.
*/
readonly attribute msmVersionedFloat AvgProductionUnit;

//------------------------------------------------------------------------
-
//    Methods
//------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    DATABASE ACCESS
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface provides the operations that control database access including
Read and Update transaction control, Commit or Rollback, and a general object
locate capability. This interface contains no data attributes and is not made
persistent in the data store. A client will create an object of this type in
order to make the persistent data store active. This object is created
automatically by the CORBA server.  The methods to create objects have been
added to this interface.
*/
interface  msmDbAccess  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

//   None defined.

//-------------------------------------------------------------------------
-
//    Exceptions
//-------------------------------------------------------------------------
-

exception  msmOpenFailed { } ;

exception  msmTransactionFailed { } ;

exception  msmWriteLocked { } ;

exception  msmCommitFailed { } ;

exception  msmNameNotFound { } ;

exception  msmInvalidObjectType { } ;

exception  msmStatusLock { } ;

exception  msmCopyNotAllowed { } ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-
```

```
/**
Initiates database.
*/
void  Opendatabase ( )
raises ( msmOpenFailed ) ;
```

```
/**
Initiates a non-blocking read transaction.
*/
void  BeginReadTrans ( )
raises ( msmTransactionFailed ) ;

/**
Ends a read transaction.
*/
void  EndReadTrans ( ) ;

/**
Initiates an update transaction.
*/
void BeginUpdateTrans ( )
raises ( msmWriteLocked ) ;

/**
Commits updates to data store.
*/
void Commit  ( )
raises (msmCommitFailed) ;

/**
Rolls back current update transaction.
*/
void Rollback ( ) ;

/**
Returns a library object of a given type.
@parm lib - the library object
@returns A library object of a given type
*/
msmLibrary  GetLibrary ( in string lib )
raises ( msmNameNotFound ) ;

/**
Returns a SimReqst object by name.
@parm nam - the SimReqst object
@returns A SimReqst object by name
*/
msmSimReqst GetSimReqst ( in string nam )
raises ( msmNameNotFound ) ;

/**
Creates an object.
@parm typ - the type of object being created
@parm nam - the name of the object
@parm desc - the description of the object
@returns An object of the specified type with a name and description
*/
msmBaseObject CreateObject ( in ObjectType typ, in string nam, in string desc
)
raises ( msmInvalidObjectType ) ;
```

```
/**
Copy an object.
@parm refObject - the object to be copied
@parm nam - the name of the object
@parm desc - the description of the object
@returns A copy of the object with a new name and description
Object types that may be copied by a client:
 Break, Breakdown, DesignAlternative, DesignStudy, InflationTable, Material,
 MfgProgram, Part, Personnel, ProcessPlan, RefProcess, SimReqst, Tool,
 WorkCalendar, WorkShift
*/
msmBaseObject CopyObject ( in msmBaseObject refObject, in string nam, in
string desc )
raises ( msmCopyNotAllowed );
} ;
```

```
//---------------------------------------------------------------------------
-
//---------------------------------------------------------------------------
-
//    DATE / TIME
//---------------------------------------------------------------------------
-
//---------------------------------------------------------------------------
-

/**
Implements a standard date-time format for this model.
*/
interface  msmDateTime : msmBaseObject  {

//---------------------------------------------------------------------------
-
//    Attributes
//---------------------------------------------------------------------------
-

/**
Returned string format: yyyy/mm/dd 24:mm:ss.
*/
readonly  attribute string  DateTime ;

//---------------------------------------------------------------------------
-
//    Exceptions
//---------------------------------------------------------------------------
-

/**
Returns year, month, day, hour, minute, or second.
*/
exception msmInvalidDateTime{string errorterm;} ;

//---------------------------------------------------------------------------
-
//    Methods
//---------------------------------------------------------------------------
-

/**
Sets value from system clock.
*/
void SetCurrentDateTime ( ) ;

/**
Format : yyyy/mm/dd 24:mm:ss
@parm DT - the date-time string
*/
void SetDateTime ( in string DT )
raises ( msmInvalidDateTime ) ;

/**
Validates a date time string.
```

```
@parm dt - the date-time string
@returns The boolean value.
*/
boolean ValidateDateTime ( in string dt ) ;
```

```
/**
Create a date time string.
@parm yr - the year,
@parm mo - the month,
@parm day - the day,
@parm hr - the hour,
@parm min - the minute,
@parm sec - the second
*/
string CreateDateTime ( in long yr, in long mo, in long day, in long hr, in
long min, in long sec )
raises ( msmInvalidDateTime ) ;

/**
Parses a date time string.
@parm dt - the date-time string,
@parm yr - the year,
@parm mo - the month,
@parm day - the day,
@parm hr - the hour,
@parm min - the minute,
@parm sec - the second
*/
void  ParseDateTime ( in string dt, out long yr, out long mo, out long day,
out long hr, out long min, out long sec )
raises ( msmInvalidDateTime ) ;
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    DESIGN STUDY ALTERNATIVE
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
A design study alternative defines one approach to a design or process
decision. Typically a design study will include a small number or
alternatives, but this model places no limit on the number of alternatives.
Constructor will create ProcessPlans sequence. User must add MfgProg after
finding one in the library or creating one.
*/
interface  msmDesignAlternative  :  msmNamedObject  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Identifies the baseline process plan.
*/
attribute  msmProcessPlan  BaselineProcPlan;

/**
Configuration management flag that sets the status of the alternative being
evaluated:  working, review, released.
*/
attribute  Status  DAStatus ;

/**
Associated mfg program information.
*/
attribute  msmMfgProgram  MfgProgram ;

/**
List of the process plan versions created for a particular alternative that
allows investigation of several plans for each alternative.
*/
attribute  msmObjectSeq ProcessPlans ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    DESIGN STUDY
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface provides a container for data associated with a manufacturing
design study. It includes the evaluation of one or more alternatives with
respect to specific variables. Design studies are the top-level data object
within this model and provide overall configuration management controlled by
the "status" attribute.  Constructor will create the alternative sequence and
set status=working. Constructor will automatically place this object in the
appropriate library.

*/
interface  msmDesignStudy  :  msmNamedObject  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Configuration management status (working, review, released).
*/
attribute Status  DSStatus ;

/**
Alternative selected from a particular design study in terms of the defined
variables (e.g., cost, schedule, risk).
*/
attribute msmDesignAlternative SelectedAlternative;

/**
URL for the web page that contains summary info for the design study.
*/
attribute string SummaryURL ;

/**
List of alternatives studied.
*/
attribute msmObjectSeq  Alternative ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//--------------------------------------------------------------------------
-
//--------------------------------------------------------------------------
-
//    FEATURE
//--------------------------------------------------------------------------
-
//--------------------------------------------------------------------------
-

/**
This interface defines a class for part features.  Can be design or
manufacturing feature depending on their use. Constructor will create cost
object and characteristics sequence.
*/
interface  msmFeature   :  msmNamedObject  {

//--------------------------------------------------------------------------
-
//    Attributes
//--------------------------------------------------------------------------
-

/**
Quantity of this feature in a given part.
*/
attribute long Quantity;

/**
Cost of this feature (total if quantity > 1). Broken into tooling, labor, and
material costs.
*/
readonly attribute msmCostInfo Cost;

/**
Feature type - may be a base part or other specific feature such as hole.
*/
attribute string Type;

/**
List of characteristics Name-Value pairs.
*/
attribute msmObjectSeq Characteristics;

//--------------------------------------------------------------------------
-
//    Methods
//--------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    INFLATION TABLE
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
This interface contains a table of inflation values for a range of years.
This is used to define the inflation assumed in forward year dollars.
Constructor will automatically place this object in the appropriate library.
*/
interface  msmInflationTable  :  msmNamedObject  {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

//   None defined.

//------------------------------------------------------------------------
-
//    Exceptions
//------------------------------------------------------------------------
-

exception  msmYearNotInTable { } ;

exception  msmYearExists { } ;

//------------------------------------------------------------------------
-
//    Methods
//------------------------------------------------------------------------
-

/**
Adds year to table - will not over write.
@parm yr - the year to add,
@parm rate - the inflation rate
*/
void AddYear ( in long yr, in float rate )
raises (msmYearExists) ;

/**
Changes data for a year.
@parm yr - the year to change,
@parm rate - the inflation rate
*/
void ChangeYear ( in long yr, in float rate )
raises ( msmYearNotInTable ) ;
```

```
/**
Returns inflation rate for a given year.
@parm yr - the year
@returns The inflation rate for a given year.
*/
float  GetRate ( in long yr )
raises ( msmYearNotInTable ) ;
```

```
/**
Sets inflation rate to use beyond table.
@parm extrap - the inflation rate
*/
void SetExtrapRate ( in float extrap ) ;

/**
Returns cost adjusted to newyear.
@parm origyear - the original year,
@parm newyear - the new year
@returns The cost adjusted to newyear.
*/
float  InflateCost ( in long origyear, in float cost, in long newyear)
raises ( msmYearNotInTable ) ;
} ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    LIBRARY
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
This interface defines a list with access methods that are used to store
references to all instances a particular type of object.  The msmDBAccess
object has a method to locate a particular Library object.  Library objects
are created when a database is initialized, although new libraries can be
created at any time.

Constructor will create the LibraryList sequence, declaring objects to be
unique.  Standard library objects are created at time CORBA server is first
built.

Note to server developers - Constructors for Library objects should add new
objects to the library.

The following Libraries are defined:

Break,
CAD Model,
Design Alternative,
Design Study,
Inflation Table,
Material,
MfgProgram,
Part,
Personnel,
ProcessPlan,
Reference Process,
Simulation Request,
Tool,
WorkCalendar,
WorkShift,
*/
interface msmLibrary :  msmNamedObject {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

/**
This is the list of related objects.
*/
attribute  msmObjectSeq  LibraryList ;

//------------------------------------------------------------------------
-
```

```
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    MATERIAL
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
This interface describes the materials that are used in producing parts.
Constructor will create characteristics sequence and unit cost. Constructor
will automatically place this object in the appropriate library.
*/
interface  msmMaterial  :  msmNamedObject  {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

/**
Unit cost in dollars per unit (ex: dollars/pound).
*/
readonly attribute msmValueWithUnits  UnitCost ;

/**
Name/Value pairs of material characteristics.
*/
attribute  msmObjectSeq  Characteristics ;

/**
Material type (e.g., aluminum, steel).
*/
attribute  string  Type ;

/**
Material form (e.g., bar, sheet).
*/
attribute  string  Form ;

//------------------------------------------------------------------------
-
//    Methods
//------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    MANUFACTURING ORDER
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
This interface represents a manufacturing order for a quantity of parts. Its
primary information is a quantity that can not be directly related to an
alternative or process plan and the planned scheduling for the order.  It
initiates the execution of a process plan on the shop floor. Constructor
creates the schedule object.  Constructor initializes name, description, and
quantity.
*/
interface  msmMfgOrder  :  msmNamedObject  {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

/**
Number of parts in an order.
*/
attribute  long  Quantity  ;

/**
Schedule information associated with the order.
*/
readonly attribute msmSchedInfo  Schedule ;

//------------------------------------------------------------------------
-
//    Methods
//------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    MANUFACTURING PROGRAM
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface describes a manufacturing program and contains information on
production quantity and build rate.  It is associated with a specific design
study. Constructor will automatically place this object in the appropriate
library.

*/
interface   msmMfgProgram  : msmNamedObject  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Production quantity for the program.
*/
attribute long ProdQty;

/**
Monthly build rate.
*/
attribute long BuildRatePerMonth;

/**
Time it takes to build first article in hours.
*/
attribute float FirstArticleTimeHr;

/**
Slope of production learning curve.
*/
attribute float LearningCurveSlope;

/**
Year Technology - Defines the technology readiness year for the program.
*/
attribute long TechYear;

/**
Average time it takes to build a unit.
*/
attribute float AvgArticleTimeHr;

//-------------------------------------------------------------------------
-
```

```
//    Methods
//------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    OBJECT SEQUENCE
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
This interface defines a list of similar objects and provides the operations
to add and access these objects. Constructor for this object will only be
called by other objects within the server that contain msmObjectSeq's.
Constructor must control whether duplicate names are allowed, and operations
must check as appropriate.  Constructor will create the object sequence and
the object struct sequence.
*/
interface   msmObjectSeq : msmBaseObject  {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

/**
Returns a boolean indicating if this sequence accepts duplicates.
*/
readonly attribute boolean DuplicatesAllowed ;

/**
Defines the type of object in the sequence.
*/
readonly attribute ObjectType SeqType ;

/**
Number of objects currently in this sequence.
*/
readonly attribute long NoInSeq ;

/**
Returns a sequence of the all objects in the sequence.
*/
attribute baseObjectSeq objectSequence ;

/**
Returns a sequence of all objects in the sequence along with their names and
descriptions.
*/
readonly attribute baseObjectStructSeq objectStructSeq ;


//------------------------------------------------------------------------
-
//    Exceptions
```

```
//------------------------------------------------------------------------
-

/**
Object specified in Find, Delete, or Insert was not found.
*/
exception  msmObjectNotFound { } ;
```

```
/**
Specified index is greater than number of objects in list.
*/
exception  msmIndexExceedsNo { } ;

/**
Duplicate object not allowed.
*/
exception  msmDuplicateObject { } ;

//------------------------------------------------------------------------
-
//    Methods
//------------------------------------------------------------------------
-

/**
Adds object to list.
@parm newObject - the object to add
*/
void  AddObject ( in msmBaseObject  newObject )
raises ( msmDuplicateObject ) ;

/**
Inserts object before ref object.
@parm newObject - the new object,
@parm refObject - the reference object
*/
void  InsertBefore( in msmBaseObject newObject, in msmBaseObject refObject )
raises ( msmDuplicateObject, msmObjectNotFound ) ;

/**
Removes object if in sequence.
@parm remObject - the object to remove from the sequence
*/
void  RemoveObject ( in msmBaseObject  remObject )
raises ( msmObjectNotFound ) ;

/**
Finds object, given its name, starting from a given index.
@parm nam - the object name,
@parm strtindx - the starting index,
@parm objindx - the object index
*/
msmBaseObject  FindObject ( in string nam, in long strtindx, out long objindx
)
raises ( msmObjectNotFound );

/**
Returns object, based on index.
@parm indx - object index
@returns The object.
*/
msmBaseObject  GetByIndex ( in long indx )
raises ( msmIndexExceedsNo ) ;
} ;
```

```
//-----------------------------------------------------------------------
-
//-----------------------------------------------------------------------
-
//    OPERATION
//-----------------------------------------------------------------------
-
//-----------------------------------------------------------------------
-

/**
This interface models an operation, or job step within a process plan. For
generality a job step may be another complete process plan. Constructor will
create Cost, Schedule, and Risk objects. Constructor will create Precedents,
Characteristics, PersonResApplic, ToolResApplic, Parts, and Features
sequences. The list of characteristics for this operation will be copied from
the reference process when it is added.
*/
interface  msmOperation  : msmNamedObject  {

//-----------------------------------------------------------------------
-
//    Attributes
//-----------------------------------------------------------------------
-

/**
Defines the type of operation.
*/
attribute string Type ;

/**
Number of repititions within this operation.
*/
attribute long Quantity ;

/**
Denotes that is operation is on the critical path in this process plan.
*/
attribute boolean  CriticalPath  ;

/**
Contains a detailed breakdown of the time elements of this operation. Summary
time information is to be stored in the Schedule attribute. Actual duration
of operation in hours, excluding setup and queue times.
*/
attribute float Runtime ;

/**
Contains a detailed breakdown of the time elements of this operation. Summary
time information is to be stored in the Schedule attribute. Duration of the
setup in hours.
*/
attribute float SetupDurationHr ;

/**
```

```
Contains a detailed breakdown of the time elements of this operation. Summary
time information is to be stored in the Schedule attribute. Time spent in
queue in hours.
*/
attribute float QueueDurationHr ;
```

```
/**
This identifies this operation as a nested process plan.  If this field is
null, this is a discrete operation.
*/
attribute   msmProcessPlan  ProcPlan ;

/**
Queue capacity - number of items necessary to fill the queue.
*/
attribute long  QueueTotalCapacity ;

/**
Average number in queue.
*/
attribute long  QueueAvgCapacity ;

/**
Cost information for the operation.
*/
readonly attribute msmCostInfo  Cost ;

/**
Schedule information for the operation.
*/
readonly attribute msmSchedInfo  Schedule ;

/**
Set of risk information associated with the operation.
*/
attribute msmObjectSeq Risk ;

/**
Adding the Reference Process causes its list of OpChar characteristics to be
added to the Characteristics sequence below. The reference process associated
with an operation that contains standard process information.
*/
attribute msmRefProcess  RefProcess ;

/**
ID number for the operation.
*/
attribute  string  Id ;

/**
Description of setup activity.
*/
attribute string SetupDescription ;

/**
List of precedent operations.  This attribute will typically be used to
communicate complicated precedence information.  Tools that can handle only
sequential operation sequences should not populate information into this
attribute.
*/
attribute msmObjectSeq Precedents ;
```

```
/**
User defined characteristics.
*/
attribute msmObjectSeq  Characteristics ;

/**
List of personnel Resource Applications used in an operation.
*/
attribute msmObjectSeq  PersonResApplic  ;

/**
List of tool Resource Applications used in an operation.
*/
attribute msmObjectSeq  ToolResApplic ;

/**
Parts consumed by this operation.  Represented as a sequence of part usages.
*/
attribute msmObjectSeq  ConsumedParts ;

/**
Parts produced by this operation.  Represented as a sequence of part usages.
*/
attribute msmObjectSeq  ProducedParts ;

/**
List of features associated with this operation.
*/
attribute msmObjectSeq  Features ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    PART
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
This interface defines a part to be manufactured.  It may be a detailed part
or an assembly. Constructor will create the Cost object as well as the
AssociatedParts and Features sequences.  Constructor will automatically place
this object in the appropriate library.
*/
interface  msmPart   : msmNamedObject  {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

/**
Rejection rate in percentage of parts that are unacceptable.
*/
attribute  long  RejectionRate ;

/**
Identifies the part type: detailed or assembly.
*/
attribute PartType  Type ;

/**
Part complexity (low, med, high).
*/
attribute  TextQual   Complexity ;

/**
Cost information for a part.
*/
readonly attribute msmCostInfo Cost ;

/**
Associated CAD models.
*/
attribute msmObjectSeq  CADModels;

/**
Part material.
*/
attribute msmMaterial  Material ;
```

```
/**
Part number.
*/
attribute string  Number ;

/**
Part family.
*/
attribute string  Family ;

/**
List of parts in assembly.
*/
attribute msmObjectSeq  AssociatedParts;

/**
List of features for a detailed part.
*/
attribute msmObjectSeq  Features ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    PART LOCATION
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface represents the location of a part as it is used in a given
operation.  It is created automatically when the quantity in the msmPartUsage
is set.  This object may not be created directly by the clients, but its
attributes may be populated by them.
*/
interface  msmPartLocation : msmNamedObject  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Transformation matrix use to map resources CAD model into application
space(typedef float Matrix [4] [4];).
*/
attribute Matrix TranformMatrix;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    PART USAGE
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
This interface defines the usage of a part.  Part Usage name should be the
same as the part number for the part defined in the usage.  When a msmPart is
added to a msmPartUsage object, the msmPartUsage name attribute is
automatically overwritten with the part number from the msmPart object.
*/

interface  msmPartUsage   : msmNamedObject  {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

/**
The number of units of a part used in a given operation.
*/
readonly attribute long Quantity ;

/**
The part for this usage.
*/
attribute msmPart  Part;

/**
The locations (transformation matrices) for the part.  The number of
locations in the sequence will be forced to match the quantity of the given
part is this part usage if the boolean in the SetQuantity method is true.
This sequence should only be created/modified through the SetQuantity method
to make sure there is consistency.
*/
attribute msmObjectSeq  PartLocations;


//------------------------------------------------------------------------
-
//    Methods
//------------------------------------------------------------------------
-

/**
Sets the quantity attribute.  If the createloc is set to true, the method
automatically creates the msmPartLocation sequence with the number in the
sequence being equal to the quantity being set.  For increases in the
quantity, empty msmPartLocation objects are added to the sequence.  For
```

```
decreases in the quantity, a new sequence is created with empty
msmPartLocation objects.
@parm newquantity - the new quantity
@parm createloc - flag for creating the msmPartLocation sequence.
*/
void SetQuantity (in long newquantity, in boolean createloc);
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    PERSONNEL
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface defines a personnel resource. It inherits from msmResource.
Constructor will automatically place this object in the appropriate library.

*/
interface  msmPersonnel   :   msmResource  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Defines resource cost - $ / Hour.
*/
attribute float LaborRate ;

/**
Year for which labor rate is effective.
*/
attribute long LaborRateYear ;

/**
Sequence of Work calendars for this person.
*/
attribute msmObjectSeq Calendars;

/**
Work shift for this person.
*/
attribute msmWorkShift Shift;

/**
Defines a personnel skill category.
*/
attribute  string  Skill ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    PROCESS PLAN
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
This is a central object within this model.  It contains an ordered set of
operations (job steps) which define a manufacturing process.  For purposes of
generality, a job step in one process plan can be another process plan.  A
process plan has an associated part (detailed or assembly) and rolls up the
cost, schedule, and risk values of its job steps. Constructor will create
Cost and Schedule. Characteristics, SimMod, Risk, ToolPool, PersonnelPool,
and Operations sequences will be created but not populated. Constructor will
automatically place this object in the appropriate library.
*/
interface  msmProcessPlan  :  msmNamedObject  {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

/**
Configuration management flag that sets the status of the process plan being
evaluated: working, review, released.  If status is working, changes may be
made to the process plan.  When a process plan status is either review or
released, all objects associated with the process plan are locked with the
exception of reference type data that is typically stored in a library.
These include break, breakdown, CAD Model, Inflation Table, Material,
Manufacturing Program, Part, Personnel, Reference Process, Tool, Work
Calendar and Work Shift.
*/
attribute  Status  PPStatus ;

/**
Average time for critical path steps, in hours.
*/
attribute float  AvgHrCriticalPath ;

/**
Time spent waiting, hours  - This attribute is simply a method to sum all
queue durations.
*/
readonly attribute float  WaitHr ;
```

```
/**
Engineering bill of materials associated with the process plan.  Typically
the starting point for a planning activity.  Associated parts for EBOM should
be identified as such in name of part to provide uniqueness.  Engineering
bill is reference data and should not be modified by simulation tools once
created.
*/
attribute  msmPart  EBOM ;

/**
Manufacturing bill of materials associated with the process plan.  Associated
parts for MBOM should be identified as such in the name of the part to
provide uniqueness.  MBOM is developed from modifications to the EBOM for
manufacturing considerations.  Low level parts or common (unchanged) are
reused and are not specific to either bill.
*/
attribute  msmPart  MBOM ;

/**
Cost information for the process plan.
*/
readonly attribute msmCostInfo  Cost ;

/**
Schedule information for the process plan.
*/
readonly attribute msmSchedInfo  Schedule ;

/**
Set of risk information associated with the process plan.
*/
attribute msmObjectSeq Risk;

/**
Identifies manufacturing orders associated with the process plan.
*/
attribute msmObjectSeq MfgOrders ;

/**
List of characteristics for this process plan.
*/
attribute msmObjectSeq Characteristics;

/**
List of associated simulation model(s).
*/
attribute msmObjectSeq SimMod ;

/**
List of Tool Pools used in process.
*/
attribute msmObjectSeq  ToolPool ;

/**
List of Personnel Pools used in process.
*/
attribute msmObjectSeq  PersonnelPool ;
```

```
/**
Ordered list of operations or job steps in the plan - An operation may be
another process plan.
*/
attribute msmObjectSeq  Operations  ;
```

```
//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    REFERENCE PROCESS
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface defines the set of characteristics about a standard
manufacturing process.  A few general attributes are explicitly defined, and
a Characteristics object is included to allow other attributes to be defined
for the reference process.  A second Characteristics object is included to
define the characteristics needed by the Operation object that is of this
Reference Process type. Constructor will automatically place this object in
the appropriate library.  Constructor will create Risk object and
Characteristics and OpChar sequences.
*/
interface  msmRefProcess  :  msmNamedObject  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Value from historical data for the standard hours it takes to complete the
process.
*/
attribute  float StdHrs ;

/**
Qualitative process stability:  low, medium, high.
*/
attribute  TextQual  Stability ;

/**
Qualitative process maturity:  low, medium, high.
*/
attribute  TextQual   Maturity ;

/**
Qualitative process complexity (0-10).
*/
attribute  NumQual   Complexity ;

/**
Set of risk data associated with the process.
*/
attribute msmObjectSeq Risk;
```

```
/**
Name/Value pairs of characteristics that apply to the reference process.
*/
attribute msmObjectSeq Characteristics;

/**
Characteristics needed by actual Operation. These will be copied into
Operation when it is defined to be of this ref process type.
*/
attribute  msmObjectSeq  OpChar ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    RESOURCE APPLICATION
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
This interface represents the utilization of a resource in one operation. It
is used to specify the location of the resource in the factory model and the
percentage of the available resource that is used in this one operation.
Constructor sets Utilization = 0.
*/
interface  msmResourceApplic : msmNamedObject  {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

/**
Quantity in this Application.
*/
readonly attribute float Quantity ;

/**
Transformation matrix use to map resources CAD model into application space
(typedef float Matrix [4]  [4];).
*/
attribute Matrix TranformMatrix;

/**
Pool from which resource is drawn.
*/
attribute msmResourcePool  Pool ;

//------------------------------------------------------------------------
-
//    Exceptions
//------------------------------------------------------------------------
-

exception  msmUtilExceedsQty { };

//------------------------------------------------------------------------
-
//    Methods
//------------------------------------------------------------------------
-
```

```
/**
Sets orig or new quantity.
@parm newquantity - the new quantity
*/
void SetQuantity (in float newquantity)
raises (msmUtilExceedsQty);
} ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    RESOURCE POOL
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
This interface defines a pool of one type of resource which will track both
the total quantity of this resource that is available to a process plan and
also will keep track of the total utilization of the resources in this pool
by the process plan. The ResouceUitilization data structure will reference
this pool to track the utilization of a resource at the operation level.
Constructor will set Utilization to 0.
*/
interface  msmResourcePool : msmNamedObject  {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

/**
Quantity in this pool.
*/
readonly attribute float Quantity;

/**
Number of individual resources currently in use from this pool.  Cannot
exceed Quantity.
*/
readonly attribute float Utilization;

/**
Resource object for this pool.
*/
attribute msmResource Resource;

//------------------------------------------------------------------------
-
//    Exceptions
//------------------------------------------------------------------------
-

exception  msmUtilExceedsQty { };

//------------------------------------------------------------------------
-
//    Methods
//------------------------------------------------------------------------
-
```

```
/**
Sets orig or new quantity.
@parm newquantity - the new quantity
*/
void SetQuantity (in float newquantity)
raises (msmUtilExceedsQty);

/**
Changes utilization - should only be called by ResourceApplication object.
@parm newutil - the new utilization
*/
void ChangeUtil (in float newutil)
raises (msmUtilExceedsQty);
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    RISK INFORMATION
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface contains the basic set of risk data used throughout the model.
All values are stored as versioned variables (msmVersionedFloat).
Constructor will create all msmVersionedFloat attributes.
*/
interface  msmRiskInfo : msmNamedObject  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Percentage of correct results produced by process.
*/
attribute float  RiskYield;

/**
Likelihood of the failure of a given activity as percentage defects.
*/
readonly attribute msmVersionedFloat FailureProbability;

/**
Consequence of the failure - provides indication of the level of importance
of a failure.
*/
readonly attribute msmVersionedFloat FailureConsequence;

/**
Mean value - assumes a normal distribution.
*/
readonly attribute msmVersionedFloat Mean;

/**
Standard deviation - measure of the dispersion of a of a frequency
distribution with respect to the mean.  Assumes a normal distribution.
*/
readonly attribute msmVersionedFloat StdDeviation;

/**
Process capability measurement.
*/
readonly attribute msmVersionedFloat Cp;
```

```
/**
Process capability measurement.
*/
readonly attribute msmVersionedFloat Cpk;

/**
Textual statement of the basis and assumptions involved in a particular risk
analysis. Not versioned.
*/
attribute string QualitativeResults;

/**
List of contributors to this risk and their percentage contribution. Not
versioned.
*/
attribute msmObjectSeq  Contributors;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    SCHEDULE INFORMATION
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface contains the basic set of schedule data used throughout the
model.  All date values are stored as versioned strings  (msmVersionedString)
in SAVE model standard date/time format  (yyyy/mm/dd 24:mm:ss).  Constructor
will create all msmVersionedFloat and msmVersionedString attributes.
*/
interface  msmSchedInfo : msmBaseObject  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Qualitative priority  - low, medium, high.
*/
attribute TextQual  Priority ;

/**
Planned duration in hours of a particular task.
*/
readonly attribute msmVersionedFloat PlannedDurationHr ;

/**
Actual duration in hours of a particular task.  Includes any setup, queue,
and execution time.  Individual elements are stored in msmOperation.
*/
readonly attribute msmVersionedFloat ActualDurationHr ;

/**
Planned start date for a particular task or event.
*/
readonly attribute msmVersionedString PlannedStartDate ;

/**
Desired completion for a particular task or event.
*/
readonly attribute msmVersionedString PlannedEndDate ;

/**
Actual start date for a particular task or event.
*/
readonly attribute msmVersionedString ActualStartDate ;
```

```
/**
Actual completion date for a particular task or event.
*/
readonly attribute msmVersionedString ActualEndDate ;

//------------------------------------------------------------------------
-
//    Methods
//------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    SIMULATION MODEL
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface provides a mechanism to store information about the
simulations performed for a specific alternative process plan.  It includes
the simulation software type and name as well as information about the
results or output files.  All input information is stored in the msmSimReqst.
*/
interface   msmSimModel : msmNamedObject   {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Category of simulation (e.g., Cost, Factory).
*/
attribute SimType  Type ;

/**
Information on associated factory model.
*/
attribute msmObjectSeq  Factory;

/**
Identifies the vendor code (with version) used for the simulation For
example: QUEST, Factor Aim, Cost Advantage, VSA, IGRIP, ASURE.
*/
attribute string SimCode;

/**
Location of output data generated by this execution - multiple datasets can
be defined in comma delimited list.
*/
attribute string  DataLocation;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    SIMULATION REQUEST
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface contains the primary entry points into the data model that
might be needed by a simulation code.  It is expected that this will be
passed to the simulators by the work flow manager. The attributes InputFiles,
InputOptions and OutputOptions can be used to pass a simulator information on
launch, input, and output options. These attributes use tab-delimited strings
to separate the options.

Typically the trade study team will create these SimReqst objects and include
a reference to them in the work flow being developed. The WFM passes this
reference to the simulator before launch. The simulator wrapper then accesses
the SimReqst object to obtain launch options and database context
information. The syntax of information in these strings will be determined by
each simulator. Constructor will automatically place this object in the
appropriate library.
*/
interface   msmSimReqst  :  msmNamedObject {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Design study.
*/
attribute   msmDesignStudy  DesignStudy ;

/**
Design study alternative.
*/
attribute   msmDesignAlternative  DesignAlternative ;

/**
Process plan.
*/
attribute   msmProcessPlan  ProcessPlan  ;

/**
Defines optional input files and/or launch instructions.
*/
attribute   string  InputFiles ;
```

```
/**
String defining which data model input options are to be exercised in this
code execution.
*/
attribute   string  InputOptions ;

/**
String defining which data model output options are to be exercised in this
code execution.
*/
attribute   string  OutputOptions ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    TOOL
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
Defines a tool type of resource. This interface inherits from msmResource.
Constructor will create Characteristics and Breakdown sequences. Constructor
will automatically place this object in the appropriate library.
*/
interface  msmTool   :   msmResource  {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

/**
Cost of tool in dollars.
*/
attribute float  Cost ;

/**
Tolerance capability for tool - Inches.
*/
attribute  float  ToleranceCap ;

/**
Tool failure rate - percentage of time.
*/
attribute float  FailureRate ;

/**
List of other characteristics in name/value pairs.
*/
attribute  msmObjectSeq  Characteristics;

/**
Type of tool (e.g., stationary or moveable).
*/
attribute  string  Type ;

/**
Breakdown parameters.
*/
attribute msmObjectSeq  Breakdowns ;
```

```
//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

```
//---------------------------------------------------------------------------
-
//---------------------------------------------------------------------------
-
//    VALUE WITH UNITS
//---------------------------------------------------------------------------
-
//---------------------------------------------------------------------------
-

/**
Initially, msmValueWithUnits will not be very functional in the current
release of the server. Calling CheckUnits will always return a Boolean true
value and Calling Convert Value will return as a float whatever is loaded
into the Value attribute.
*/
interface msmValueWithUnits  : msmBaseObject  {

//---------------------------------------------------------------------------
-
//    Attributes
//---------------------------------------------------------------------------
-

/**
Attribute value.
*/
attribute float Value ;

/**
Attribute units.
*/
attribute string  Units ;

//---------------------------------------------------------------------------
-
//    Exceptions
//---------------------------------------------------------------------------
-

exception  msmInvalidUnits { } ;

//---------------------------------------------------------------------------
-
//    Methods
//---------------------------------------------------------------------------
-

/**
Validates newunits string, then converts and returns converted value.
@parm newunits - the newunits string
@returns The converted value.
*/
float  ConvertValue ( in string newunits )
raises ( msmInvalidUnits ) ;
```

```
/**
Validates a unit string as being defined in conversion routines.
@parm - the unit string
@returns The boolean true value.
*/
boolean CheckUnits ( in string unit )
raises ( msmInvalidUnits ) ;
} ;
```

```
//-----------------------------------------------------------------------
-
//-----------------------------------------------------------------------
-
//    VERSIONED FLOAT
//-----------------------------------------------------------------------
-
//-----------------------------------------------------------------------
-

/**
This interface implements a fine-grained versioning for variables of type
float.  All versions of a value are retained along with Date, Source, and
Quality of estimate values.  A readonly attribute is provided for easy access
to the current value.  Previous values can be retrieved by date with the
getDatedValue method.
*/
interface  msmVersionedFloat : msmBaseObject  {

//-----------------------------------------------------------------------
-
//    Attributes
//-----------------------------------------------------------------------
-

/**
Provides a sequence of the versioned floats for a particular attribute along
with its value, source, and datetime.
*/
readonly  attribute  VersFloatStructSeq VersionedFloatStructSeq ;

/**
Current numeric value.
*/
readonly  attribute  float  CurrValue ;

/**
Code/version which generated current data.
*/
readonly  attribute  string  CurrSource ;

/**
Date/Time when current value was created.
*/
readonly attribute  string  CurrDateTime ;

//-----------------------------------------------------------------------
-
//    Exceptions
//-----------------------------------------------------------------------
-

exception  msmNoValidValue { } ;
```

```
//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

/**
Adds a new value and sets CurrDateTime.
@parm val - the new value,
@parm srce - the source
*/
void  addNewValue  ( in float  val,  in string  srce );

/**
Returns a value closest to, but prior to, the datetime specified.
@parm dattim - the date-time string
@returns A value based on a date-time requirement.
*/
float getDatedValue ( in string  dattim )
raises ( msmNoValidValue ) ;

/**
Returns information based on a datetime requirement.
@parm dattim - the date-time string,
@parm srce - the source
@returns Information based on a date-time requirement.
*/
float getDatedInfo  ( in string dattim,  out string srce )
raises ( msmNoValidValue ) ;
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    VERSIONED STRING
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface implements a fine-grained versioning for variables of type
string.  All versions of a value are retained along with Date, Source, and
Quality of estimate values. A readonly attribute is provided for easy access
to the current value. Previous values can be retrieved by date.
*/
interface  msmVersionedString : msmBaseObject  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Provides a sequence of the versioned strings for a particular attribute along
with its value, source, and datetime.
.
*/
readonly  attribute  VersStringStructSeq VersionedStringStructSeq ;

/**
Current value.
*/
readonly  attribute  string  CurrValue ;

/**
Code / versions which generated current data.
*/
readonly  attribute  string  CurrSource ;

/**
Date/Time when current value was created.
*/
readonly attribute   string  CurrDateTime ;

//-------------------------------------------------------------------------
-
//    Exceptions
//-------------------------------------------------------------------------
-

exception  msmNoValidValue { } ;
```

```
//------------------------------------------------------------------------
-
//    Methods
//------------------------------------------------------------------------
-

/**
Adds a new value and sets CurrDateTime.
@parm val - the new value,
@parm srce - the source
*/
void  addNewValue  ( in string  val,  in string  srce );

/**
Returns a value closest to, but prior to, the datetime specified.
@parm dattim - the date-time string
@returns A value based on a date-time requirement.
*/
string  getDatedValue ( in string  dattim )
raises ( msmNoValidValue ) ;

/**
Gets information based on a datetime requirement.
@parm dattim - the date-time string,
@parm srce - the source
@returns Information based on a date-time requirement.
*/
string  getDatedInfo  ( in string dattim,  out string srce )
raises ( msmNoValidValue ) ;
} ;
```

```
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-
//    WORK CALENDAR
//------------------------------------------------------------------------
-
//------------------------------------------------------------------------
-

/**
This interface defines the calendar of available work days for a resource.
Constructor will automatically place this object in the appropriate library.
*/
interface  msmWorkCalendar   :  msmNamedObject  {

//------------------------------------------------------------------------
-
//    Attributes
//------------------------------------------------------------------------
-

/**
Year for work calendar.
*/
attribute long WorkYear ;

/**
Day of week on which Jan 1 falls.
*/
readonly attribute DaysOfWeek  Jan1 ;

/**
Number of days which are not weekend or holidays.
*/
readonly attribute long  NoWorkDays ;

//------------------------------------------------------------------------
-
//    Methods
//------------------------------------------------------------------------
-

/**
Makes date a work day - date is valid DateTime string.
@parm date - the work day to add
*/
void  AddDay ( in string date ) ;

/**
Makes day an off day - date is valid DateTime string.
@parm date - the off day to add
*/
void  DelDay ( in string date ) ;
```

```
/**
Increment between 2 years must be done in 2 different calls. Returns number
of
workdays between dates.
@parm strt - the start date,
@parm stop - the stop date
*/
long  WorkDaysBetweenDates ( in string strt, in string stop ) ;

/**
Date is standard datetime string - Year and time ignored.
@parm date - the date
@returns The boolean value.
*/
boolean IsWorkDay ( in string date ) ;
} ;
```

```
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-
//    WORK SHIFT
//-------------------------------------------------------------------------
-
//-------------------------------------------------------------------------
-

/**
This interface defines a working shift to be used in work calendars.
Constructor creates Break sequence. Constructor will automatically place this
object in the appropriate library.
*/
interface  msmWorkShift   :  msmNamedObject  {

//-------------------------------------------------------------------------
-
//    Attributes
//-------------------------------------------------------------------------
-

/**
Start time - hours after midnight (0-24).
*/
attribute  float  StartTime ;

/**
End time - hours after midnight(0-24).
*/
attribute  float  EndTime ;

/**
List of breaks associated with this workshift.
*/
attribute  msmObjectSeq  BreakList ;

/**
Number of productive hours in shift.
*/
readonly attribute float HrInShift ;

//-------------------------------------------------------------------------
-
//    Methods
//-------------------------------------------------------------------------
-

//   None defined.
} ;
```

2.0 SAVE Work Flow Manager

```
#ifndef Simulator_idl
#define Simulator_idl

/*
Simulation Assessment Validation Environment (SAVE)
Simulator IDL
Version 2.0
October 15, 1998

Distribution Statement A:
Approved for public release; Distribution is unlimited.
*/

/**
The simulator module defines the interfaces for remote simulation
applications. A simulator has properties, operations and state. There are
methods to set and get the properties and operations and invoke the
operations with command strings. Simulator state changes generate events that
call methods on their registered listener objects.
*/

/*
IDL Changes
In order to register the same server multiple times in the WFM, a method to
add an ID to a simulator as it registers the listeners is provided.  When the
WFM wants to send an operation to a specific simulator, it supplies the
appropriate listener ID.

A simulator has a single method for invoking operations. The method arguments
consist of an enumerated simulator operation and a command string for that
type of operation. Although there are a set number of operations for all
simulators, there are an unlimited number of commands for any given
operation. An operation does not have to support any commands, which is
indicated by an empty (not a null) command string. There are methods to get
the available operations and commands so a user may select the ones required
for a task. Exceptions are raised if there is an attempt to execute an
invalid operation or command or the operation fails.

A simulator is initialized in several steps. The steps and their simulator
methods are:

getOperations() - Get the operations and see if the PREPARE operation is
among them. If there is no PREPARE, there is no initialization required for
the simulator. If it is known the simulator must be initialized, this step
may be skipped.

getProperties() - Get the properties for manual entry. If the properties come
from a data source, this step is usually skipped.

setProperties() - Set the properties of the simulator.

getCommands(SimulatorModule.PREPARE) - Get the commands for the PREPARE
operation. If the commands are already known, this step may be skipped.
```

doOperation(SimulatorModule.PREPARE, command_string) - Invoke the PREPARE command with the appropriate command string. If the initialization fails, an OperationFailed exception is raised with the reason inside it.

Simulators communicate events with other objects using listeners. A listener is a remote object that registers itself with a simulator, thus indicating it wishes to be informed of any events. As a simulator runs, it keeps its listeners appraised of its progress by invoking methods on them.

A listener is a light-weight object created by a process and registered with a simulator. It is up to the listener implementation to notify its parent process about the event. Listener objects are transient by nature and a process must explicitly remove them from a simulator before destroying the listener. There are several listener events available from the simulator.

One listener event is a notification about a state change during the normal course of operation. This is normally invoked by the simulator. Another type of state event is a user notification of a state change caused by the simulator operator. The difference between the two is the simulator notification does not require an acknowledgment while the operator notification expects an acknowledgment. It is up to the listeners to decide if they want to send back an acknowledgment for the second type of event. In general, a listener that is actively controlling a simulator would send back an acknowledgment while a listener that is only monitoring the simulator would not return an acknowledgment.

There are two types of listener messages. The first is the generic message that contains any type of information such as status, warnings and errors. The other contains the script instructions for a human operator. When a simulator receives a script, it notifies all listeners and passes them the script string. In addition, if a new listener registers after the receipt of a script string, the simulator must immediately notify the new listener about the script string.

Since there is no way for a listener to know which simulator invoked an event, the simulator supplies its CORBA name as one of the parameters in each method call. Where a process's listener is only registered with one simulator, the process can ignore the name since it only knows about one simulator. If a process's listener is registered with several simulators, however, the process uses the simulator's name to determine which simulator generated the event.

There are properties that define a simulator's context. These properties are set before invoking the PREPARE operation. There are accessor methods to get and set the properties. The setProperties() method may include only a partial list of properties. Properties are specified by a name, description, string value and enumerated type. When setting properties, exceptions are raised if a property does not exist or its value is invalid (i.e. cannot be converted to the appropriate type) and none of the properties are set. This behavior allows validating all properties before setting any of them.

```
In order to keep the Simulator interface separate from the main SAVE IDL, the
Simulator object does not inherit from msmNamedObject like the other objects
in SAVE. Since a simulator is not a library, this has no impact on the rest
of the system. For now, the name given to the simulator is the CORBA name of
the server; that is, the one returned by a call to object_name() on the CORBA
object.
*/
```

```
//---------------------------------------------------------------------------
//    Enumerations
//---------------------------------------------------------------------------

/**
The property type enumeration. An OCTET_TYPE is a byte while an OBJREF_TYPE
means the value string is the CORBA name of an object.
*/
enum PropertyType { OCTET_TYPE, SHORT_TYPE, LONG_TYPE, FLOAT_TYPE,
                    DOUBLE_TYPE, STRING_TYPE, OBJREF_TYPE };

/**
The simulator state enumerations. These are returned to any registered
listeners or by the getStatus() method.
*/
enum SimulatorState { UNDEFINED, INITIALIZED, ENABLED, WORKING, COMPLETED,
PAUSED, RESUMED, TERMINATED, FAULTY };

/**
The simulator operation enumerations. A simulator does not have to support
all the operations. The PREPARE is the simulator initialization and is always
the first operation done after setting the properties.
*/
enum SimulatorOperation { PREPARE, LAUNCH, TERMINATE, PAUSE, RESUME };
```

```
//--------------------------------------------------------------------------
//    Structures
//--------------------------------------------------------------------------

/**
A structure representing a single attribute of a simulator. It contains the
property's name, description and the value as a string. The type represented
by the string is defined by the enumerated PropertyType entry.
*/
struct Property {
    string name;
    string description;
    string value;
    PropertyType type;
};

/**
Contains the platform information about the simulator. It contains the host
name, IP address and operating system.
*/
struct Platform {
    string hostName;
    string hostIPAddr;
    string operSys;
};
```

```
//-----------------------------------------------------------------------------
//    Type Definitions
//-----------------------------------------------------------------------------

/**
Defines a sequence of strings.
*/
typedef sequence<string> StringSeq;

/**
Defines a sequence of properties.
*/
typedef sequence<Property> PropertySeq;

/**
Defines a sequence of simulator operations.
*/
typedef sequence<SimulatorOperation> SimulatorOperationSeq;
```

```
//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//    SIMULATOR LISTENER
//-------------------------------------------------------------------------
//-------------------------------------------------------------------------

/**
The simulator listener contains methods for notifying the listener when the
simulator changes state or there is a message.
*/
interface SimulatorListener {

//-------------------------------------------------------------------------
//    Methods
//-------------------------------------------------------------------------

/**
Invoked by the simulator when it changes state.
@parm state - the state of the simulator,
@parm name - the name of the simulator invoking the listener
*/
oneway void simulatorChangedState(in SimulatorState state, in string name);

/**
Invoked by the simulator when a user changes its state.
@parm state - the state of the simulator,
@parm name - the name of the simulator invoking the listener
*/
oneway void userChangedState(in SimulatorState state, in string name);

/**
Invoked by the simulator when there is a message.
@parm message - the simulator message,
@parm name - the name of the simulator invoking the listener
*/
oneway void message(in string message, in string name);

/**
Invoked by the simulator when it receives an instruction script.
@parm script - the instruction script,
@parm name - the name of the simulator invoking the listener
*/
oneway void script(in string script, in string name);
};
```

```
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//    SIMULATOR
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------

/**
A simulator is a CORBA server that performs operations based on the
properties supplied to it. It supports listeners, which are objects that are
notified when the simulator's state changes. A simulator is requested to
perform operations with commands specified as strings.
*/
interface Simulator {

//---------------------------------------------------------------------------
//    Exceptions
//---------------------------------------------------------------------------

/**
Exception raised if no such property exists. The name of the bad property is
returned in the string.
*/
exception NoSuchProperty { string badProperty; };

/**
Exception raised a property value does not match its type (i.e. the
conversion fails). The name of the invalid property is returned in the
string.
*/
exception InvalidProperty { string invalidProperty; };

/**
Exception raised if an operation is not available.
*/
exception NoSuchOperation { };

/**
Exception raised if a simulator is unable to support any more listeners.
*/
exception ListenersAtMaximum { };
```

```
//-------------------------------------------------------------------------
//    Methods
//-------------------------------------------------------------------------

/**
Acknowledgment for a userChangedState() event. It is up to each listener to
determine if it should invoke this method.
@parm state - the simulator state in the original event
*/
void ackUserChangedState(in SimulatorState state);

/**
Sets the properties defined in the sequence. Raises exceptions if a property
does not exist or has invalid data and does not set any of the properties in
the input sequence.
@parm properties - sequence of property objects
*/
void setProperties(in PropertySeq properties)
raises (NoSuchProperty, InvalidProperty);

/**
Gets a sequence of the available properties. Returns the sequence of
properties.
@returns The sequence of properties.
*/
PropertySeq getProperties();

/**
Returns the host and operating system information. Returns the platform
information.
@returns The Platform information.
*/
Platform getPlatform();

/**
Get the URL for the simulator's documentation. Returns the documentation URL
string. If there is no documentation, the returned string is empty.
@returns The documentation URL string.
*/
string getDocumentation();

/**
Get operations supported by this simulator. Returns the supported operations.
If there are no operations, the sequence is empty.
@returns The supported operations.
*/
SimulatorOperationSeq getOperations();
```

```
/**
Get the supported commands for the specified operation. It raises an
exception if the operation is not supported. Returns the supported commands
for the specified operation. If there are no commands for the operation, the
sequence is empty.
@parm operation - the operation
@returns The supported commands for the specified operation.
*/
StringSeq getCommands(in SimulatorOperation operation)
raises (NoSuchOperation);

/**
Performs the specified operation using the supplied command string. If the
operation does not require a command, the string is empty but not null.
Since this is a oneway method (the client does not wait around for
completion), there are no user exceptions.  The server can take as long as
necessary to perform the operation without hanging the client, provided it
still issues the required state changes to its registered listeners.  If the
operation or command is  invalid, the server should immediately change its
state to FAULTY and inform the registered listeners.
@parm operation - the operation enumeration,
@parm command - the command string
*/
oneway void doOperation(in SimulatorOperation operation, in string command,
in long listenerID)

/**
Get the current state of the simulator. This method should not block on the
simulator end but should return immediately. Returns the state of the
simulator.
@returns The state of the simulator.
*/
SimulatorState getState();

/**
Add a simulator listener to this simulator and return a unique listener ID.
If the listener is already registered, nothing changes and the ID returned is
zero. Throws an exception if the simulator is unable to support any more
listeners.
@parm listener - the simulator listener to register
@ returns The ID for the listener.
*/
long addSimulatorIDListener(in SimulatorListener listener)
raises (ListenersAtMaximum);

/**
Add a simulator listener to this simulator. If the listener is already
registered, nothing changes. Throws an exception if the simulator is unable
to support any more listeners.
@parm listener - the simulator listener to register
*/
void addSimulatorListener(in SimulatorListener listener)
raises (ListenersAtMaximum);
```

```
/**
Remove a simulator listener from this simulator. If there is no such listener
registered, nothing happens.
@parm listener - the simulator listener to unregister
*/
void removeSimulatorListener(in SimulatorListener listener);
};

#endif // Simulator_idl
```